Introduction to Compiler Design by Prof. Yi-Ping You

Project Assignment 3

Semantic Definitions

Due Date: 23:59, December 23, 2016

Your assignment is to write an LALR(1) parser for the C− (C minus) language. In the previous project, you have done syntactic definition and created a parser using **yacc**. In this assignment you will perform some simple checking of semantic correctness. Code generation will be performed in the last phase of the project.

1 Assignment

You first need to write your symbol table, which should be able to perform the following tasks:

- Push a symbol table when entering a scope and pop it when exiting the scope.
- Insert entries for variables, constants, procedure, and function declarations/definitions.
- Lookup entries in the symbol table.

You then must extend your LALR(1) grammar using **yacc**. You need to write syntax-directed translation schemes (SDTs) following the semantic definitions in the following sections.

1.1 What to Submit

You should submit the following items:

- revised version of your lex scanner
- revised version of your yacc parser
- report: a file describing what changes you have to make to your scanner/parser since the previous version you turned in, the abilities of your parser, the platform to run your parser and how to run your parser.
- a Makefile in which the name of the output executable file must be named 'parser' (**Please make sure that it works well. TAs will build your parser with this makefile. No further grading will be made if the** *make* **process fails or the executable** *parser* **is not found.**)

1.2 Pragma Directives

In the first assignemnt, we have defined:

`#pragma source`	`#pragma source on` turns on source program listing, and `#pragma source off` turns it off.
`#pragma token`	`#pragma token on` turns on token (which will be returned to the parser) listing, and `#pragma source off` turns it off.
`#pragma statistic`	`#pragma statistic on` turns on identifier frequencies listing on, and `#pragma statistic off` turns it off.

One more option will be added:

`#pragma symbol` `#pragma symbol on` dumps the contents of the symbol table associated with a block when exiting from that block, and `#pragma symbol off` turns it off.

The format of symbol table information is defined in Section 3.2.

Note that the default of this four #pragma directives are off in this assignment.

1.3 Implementation Notes

To perform semantic check, you should maintain a symbol table per scope in your parser. Each entry of a symbol table is an identifier associated with its attributes, such as its name, name type (function name, parameter name, variable name, or constant name), scope (local or global), type (variable or constant's data type, function's parameter types or function's return type), constant's value, etc. In effect, the role of a symbol table is to pass information from declarations/definitions to uses. A semantic action "puts" information about identifier x into the symbol table when the declaration of x is analyzed. Subsequently, a semantic action associated with a production such as $factor \rightarrow \textbf{id}$ "gets" information about the identifier from the symbol table.

2 Semantic Definitions

2.1 Program Units

Function

- There should be at least one function definition existed in the program.
- The function which is firstly defined is treated as the entry point of the program execution.
- A function's declaration, if declared, must appear before its definition, and the definition must match its declaration. (Once a function is declared, it must be defined somewhere after the declaration.)
- A function can be declared or defined only once.
- A function should be declared or defined before it is invoked. For example,

```
void f1(); // declaration of f1
void f2(){ // definition of f2
    f1();  // invoke f1
}
void f1(){ // definition of f1
    f2();  // invoke f2
}
```

is legal, and

```
void f0(){ // definition of f0
    // ...
}
void f1(){ // definition of f1
    // ...
}
void f2(){ // definition of f2
    f1();  // invoke f1
```

```
}
```

is also legal, but

```
void f2(){ // definition of f2
    f1();  // invoke f1, error: unknown function f1
}
```

is illegal.

- The parameter passing mechanism is call-by-value.
- Parameters of a function can be either a scalar or array type, while the return value must be a scalar type. For example,

```
int foo(int a[4]){
    return a[0];
}
```

is legal.

- The type of the return statement inside the function must match the return type of the function declaration/definition, subject to type promotion described in Section 2.4.
- For functions whose return type is not declared as void, its last statement must be a return statement. Early returns are also allowed. For example:

```
int foo(int a){
    if (a == 0) return 5; // early return
    else if (a == 1) return 10;
    return 15; // the last statement of must be a return statement
}
```

2.2 Scope Rules, Variable/Constant Declarations, Initialization

- Scope rules are similar to C.
- Names must be unique within a given scope. Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is hidden within the inner scope.
- Re-assignments to constants are not allowed, and constants cannot be declared in terms of other named constants.
- In an array declaration, the index must be greater than zero.
- For variable or array initializations, the type of the left-hand side must be the same as that of the right-hand (after performing type coercion).
- For array initializations, the number of initializers should be equal to or less than the array size. If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to 0. For example,

```
void foo{
  int a[3] = {1,2,3}; // legal
  int b[4] = {1,2,3,4,5}; // illegal
  int c[2][3] = {1,2,3,4,5,6,7}; // illegal
  int d[5] = {1}; // legal
  int e = "This is a string"; // illegal
}
```

2.3 Variable References, Array References and Expressions

- A variable reference is either an identifier reference or an array reference.
- The index of array references must be an integer type whose value is equal to or greater than zero. Bound checks are not necessary.
- Two arrays are considered to be the same type if they have the same number of elements. More specifically, they have the same number of dimensions and the same size for each dimension and the type of base element are the same.
- Array can be used in parameter passing. However, array arithmetic and assignment are not allowed. For example,

```
int foo(int a[3][3], int b[5][3], int i) {
    a[0][0] = i;       // legal
    i = a[1][1];       // legal
    a[0][1] = b[1][2]; // legal
    int k = a[3][i];   // legal : note that bounds are not checked
    i = 3+a[0];        // illegal : array arithmetic

    a = b;             // illegal : array assignment
    a[1] = b[2];       // illegal : array assignment
    return a[0][0];    // legal : 'a[0][0]' is a scalar type, but 'a' is an array type.
}
-------------------------------------------------------------------------------------
int f1(int a[5]);
int f2(int a);
int f3(float a[4]);
void f4(int x, int z[5][5][5], int t[4]) {
    x = f1(z[0][1]);    // legal
    x = f1(t);          // illegal : size mismatch
    x = f1(z[0][0][1]); // illegal
    x = f2(z[0][0][1]); // legal
    x = f3(t);          // legal : type coercion (see below)
}
```

- For arithmetic operators (`+`, `-`, `*`, or `/`), the operands' types must be either integer, float or double, and the operation produces an integer, float or double value. The resulted type is the "most general" type of the two operands. Note that an operand's type may need to be promoted to match the operand subject to the type coercion rules defined in Section 2.4.
- For the modulo operator `%`, both operands must be integers, and the operation produces an integer value.

- For logical operators (**&&**, ||, or **!**), the operands must be booleans, and the operation produce a Boolean value.
- For relational operators (`<`, `<=`, `>=`, `>`, `!=`, `==`), the operands must be integer, float or double types, and the operation produces a Boolean value. For equality comparison operators (`==` or `!=`), they must additionally support booleans as operand types. The operands can also be promoted according to the coercion rules defined in Section 2.4.
- There are no operators that operate on String values, and thus they can only appear in assignment operations, variable declarations, print statements, read statements, function arguments, or return statements. For example,

```
string a, b;
a = "compiler"; // legal
b = a;          // legal
b = b + a;      // illegal
```

2.4 Type Coercion

- An integer (float) type can be implicitly promoted into a float/double (double) type anywhere the latter is expected. For example:

```
void f1(float a);
void f2(double a[10]);
int f3();
double f3(int a, float b, double c, int d[10]) {
    b = a; // a is converted from int to float
    c = a; // a is converted from int to double
    c = b; // b is converted from float to double

    c = a + b; // a is converted to float before addition,
               // which produces a float value that is converted to double.

    f1(a); // legal, a is converted from int to float
    c = f3(); // legal, the return type int is converted to double
    f2(d); // legal, d is converted from an array of 10 int's to an array of 10 double's

    a = c; // illegal, a double value cannot be converted to an integer
    while (b == c) {} // legal, b is converted to double before comparison

    return a; // legal, the return value is implicitly converted to double
}
```

2.5 Statements

There are seven distinct types of statements: compound, simple, conditional, while, for, jump, and procedure call.

2.5.1 compound

- A compound statement forms an inner scope. Note that declarations inside a compound statement are local to the statements in the block and no longer exist after the block exits.

2.5.2 simple

- Variable references in **print** and **read** statements must be scalar type.
- In assignment statements, the type of the left-hand side must be the same as that of the right-hand side (after performing type coercion).

2.5.3 conditional and while

- The conditional expression part of **if** and **while** statements must be Boolean types.

2.5.4 for

- There are three components of a **for** loop: *initial_expression*, *control_expression* and *increment_expression*. Variables used in the above three component should be declared before the **for** statement. The control expression must be of boolean type.

2.5.5 jump

- **break** and **continue** can only appear in loop statements.

2.5.6 function invocation

- A procedure is a function that has no return value.
- The types of the actual parameters must be identical to the formal parameters in the function declaration/definition.
- The number of the actual parameters must be identical to the function declaration/definition.
- The type of the return value must be identical to the types of the return type in function declaration/definition.
- One may discard the return value of a function call. For example,

```
int f1(){
  return 5;
}
int f2(){
  f1();
}
```

2.6 Identifier

Only the first 32 characters are significant. That is, the additional part of an identifer will be discarded by the parser.

3 What Should Your Parser Do?

3.1 Error Detection

The parser should have all the abilities required in the previous project assignment and also the semantic checking ability in this project assignment. If the input is syntactically and semantically correct, output the following message.

```
|-----------------------------------------|
| There is no syntactic and semantic error! |
|-----------------------------------------|
```

Once your parser encounters a semantic error, it should output an error message containing the line number of the error and an **ERROR MESSAGE** describing the error. You are free to define your own format for **ERROR MESSAGE**, however, the whole line must be in the format of:

```
##########Error at Line #N: ERROR MESSAGE.##########
```

Notice that semantic errors should **not** cause the parser to stop its execution. You should let the parser keep working on finding as many semantic errors as possible.

3.2 More About Symbol Table

Each entry of a symbol table consists of the name, kind, scope level, type, value, and additional attributes of a symbol. Precise definitions of each symbol table entry are as follows.

Name	The name of the symbol. Each symbol have the length between 1 to 32.
Kind	The name type of the symbol. There are five kinds of symbols: function, parameter, variable, and constant.
Level	The scope level of the symbol. 0 represents global scope, local scope starts from 1, 2, 3, ...
Type	The type of the symbol. Each symbol is of types int, float, double, bool, string, or the signature of an array. (Note that this field can be used for the return type of a function)
Attribute	Other attributes of the symbol, such as the value of a constant, list of the types of the formal parameters of a function, etc.

For example, given the input:

```
 1: #pragma symbol on
 2: bool func(int a, float b);
 3: const int d = 1;
 4: void main(){
 5:     int x;
 6:     int y[10];
 7:
 8:     if(y[0] == 0){
 9:         float x;      //outer 'x' has been hidden in this scope
10:         double z;
11:         z = 0.1;
12:     }
13: }
14:
15: bool func(int a, float b){
16:     string c = "hello world";
17:     return (b <= 1.0);
18: }
```

After parsing the compound statement in function `main` (at line 12), you should output the symbol table with the following format.

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
x       variable     2(local)   float
z       variable     2(local)   double
```

After parsing the definition of function `main` (at line 13), you should output the symbol table with the following format.

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
x       variable     1(local)   int
y       variable     1(local)   int[10]
```

After parsing the definition of function `func`(at line 18), you should output the symbol table with the following format.

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
a       parameter    1(local)   int
b       parameter    1(local)   float
c       variable     1(local)   string
```

After parsing the end of the program (at line 18), you should output the symbol table with the following format.

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
func    function     0(global)  bool         int,float
d       constant     0(global)  int          1
main    function     0(global)  void
```

Here is the template for outputing symbol table.

```
printf("================================================================================================\n");
 // Name [29 blanks] Kind [7 blanks] Level [7 blank] Type [15 blanks] Attribute [15 blanks]
printf("Name                              Kind       Level       Type               Attribute   \n");
printf("------------------------------------------------------------------------------------------------\n");
printf{"a                                 function   0(global)   int                int[2],float\n");
     // ....
     // Format of Attribute: type,type,type,...
printf("================================================================================================\n");
```

4 How to Submit the Assignment?

Create a directory, named "YourID" and store all files of the assignment under the directory. Zip the directory as a single archive, name the archive as "YourID.zip". Upload the zipped file to the **e-Campus (E3)** system.

Note that the penalty for late homework is **15% per day** (weekends count as 1 day). Late homework will not be accepted after sample codes have been posted. In addition, homework assignments must be individual work. If I detect what I consider to be intentional plagiarism in any assignment, the assignment will receive **zero credit**.