Introduction to Compiler Design by Prof. Yi-Ping You

Project Assignment 3

Semantic Definitions

Due Date: 23:59, December 25, 2015

Your assignment is to write an LALR(1) parser for the $\mathcal{P}$ language. In the previous project, you have done syntactic definition and create a parser using **yacc**. In this assignment you will do 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 program declarations.
- 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 it works well. TAs will rebuild 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 Pseudocomments

In the first assignemnt, we have defined:

`S` `&S+` turns on source program listing, and `&S-` turns it off.

`T` `&T+` turns on token (which will be returned to the parser) listing, and `&T-` turns it off.

One more option will be added:

`D` Dump the contents of the symbol table associated with a block when exiting from that block if D is on (i.e. `&D+`).

The format of symbol table information is defined in Section 3.

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 (program name, 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 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

The two program units are the *program* and the *functions*.

Program

- The identifier after the **end** of a program declaration must be the same identifier as the name given at the beginning of the declaration. In addition, it must be the same as the file name. For example, for input file "test.p", the identifier at the beginning and the end of the program declaration must be "test".
- Program has no return value, thus any return statement appeared in the main block of program is not legal.

Function

- The identifier after the **end** of a function declaration must be the same identifier as the name given at the beginning of the declaration.
- The parameter passing mechanism is call-by-value.
- Parameters could be a scalar or array type, but the return value could only be a scalar type.
- The type of the return statement inside the function must be the same as the return type of the function declaration.

2.2 Scope Rules, Variable/Constant Declarations and References

- 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.
- Assignments to constants are not allowed, and constants cannot be declared in terms of other named constants.
- In an array declaration, the index of the lower bound must be smaller than that of the upper bound. Both of the indexes must be greater than or equal to zero.

2.3 Variable References, Array References and Expressions

- A variable reference is either an identifier reference or an array reference.
- Each index of array references must be an integer. Bounds checking is not performed at compile time as in C language.
- 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 types of elements are the same.
- Array types can be used in parameter passing. However, array arithmetic is not allowed. Notice that a function cannot return an array type, but it can return an element of an array. For example,

```
foo(): integer;
begin
    var a: array 1 to 3 of array 1 to 3 of integer;
    var b: array 1 to 5 of array 1 to 3 of integer;
    var i, j: integer;
    a[1][1] = i;        // legal
    i = a[1][1] + j;    // legal
    a[1][1] = b[1][2];  // legal

    a = b;              // illegal: array arithmetic
    a[1] = b[2];        // illegal: array arithmetic

    return a[1][1];     // legal: 'a[1][1]' is a scalar type, but 'a' is an array type.
end
```

- For an arithmetic operotor (`+`, `-`, `*`, or `/`), the operands must be integer or real types, and the operation produces an integer or real value. The type of the operands of an operation may be different. Please see Section 2.4 for more details.
- For a `mod` operator, the operands must be integer types, and it produces an integer value.
- For a Boolean operator (**and**, **or**, or **not**), the operands must be Boolean types, and the operation produces only Boolean value.
- For a relational operator (`<`, `<=`, `=`, `>=`, `>`, or `<>`), the operands must be integer or real types, and the operation produces only Boolean value. Operands must be of the same type.
- String operands can only appear in"`+`" operations (string concatenations), assignment statements, print statements and read statements. Notice that when doing string concatenation, both operands must be string types.

2.4 Type Coercion

- An integer type can be implicit converted into a real type due to several situations, such as assignments, parameter passing, relational expressions, or arithmetic expressions.
- The result of an arithmetic operation will be real type if at least one of the operands is real type. For example, 1.2+1, 3-1.2, and 1.2*3.4 are all result in real type.

2.5 Statements

There are seven distinct types of statements: compound, simple, conditional, while, for, return, 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 is exited.

2.5.2 simple

- Variable references of **print** or **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 unless type coercion is permitted.

2.5.3 conditional and while

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

2.5.4 for

- A counting iterative control statement has a variable, called the **loop variable**, in which the count value is maintained. The scope of the loop variable is the range of the loop. The variable is implicitly declared at the **for** statement and implicitly undeclared after loop termination.
- The value of the loop variable cannot be changed inside the loop.
- In a nested loop, each loop must maintain a distinct loop variable.
- The **loop parameters** used to compute an iteration count must be in the incremental order and must be greater than or equal to zero. For example,

  ```
  for i := 3 to 5 do
  // statements
  end do
  ```

  is a legal **for** statement, but

  ```
  for i := 5 to 3 do
  // statements
  end do
  ```

  is illegal.

2.5.5 function invocation

- A procedure is a function that has no return value.
- The types of the actual parameters and the return value must be identcial to the types of the formal parameters and the return type in the function declaration.

2.6 identifier

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?

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 the parser encounters a semantic error, output an error message, which contains the line number and the category of the error. For example,

```
<Error> found in Line 2: variable 'a' redeclared
```

Notice that semantic errors should not cause the parser to stop its execution. You should let the parser keep working on finding semantic errors as much as possible. An error should be reported with the following format:

```
char error_msg[] = "variable 'a' redeclared";
// Error messages are defined on your own according to the given semantic definitions.
printf("<Error> found in Line %d: %s\n", linenum, error_msg);
```

Each entry of a symbol table consists of the name, kind, scope level, type, value, and additional attributes of a symbol. Symbols are placed in the symbol table in order of their appearance in the input file. 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: program, 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 integer, real, boolean, 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: test;
 2:
 3: // no global declaration(s)
 4:
 5: func( a:integer ; b:array 1 to 2 of array 2 to 4 of real ): boolean;
 6: begin
 7:         var c: "hello world!";
 8:         begin
 9:                 var d: real;
10:                 return (b[1][4] >= 1.0);
11:         end
12: end
13: end func
```

```
14:
15: begin
16:         var a: integer;
17:         begin
18:                 var a: boolean;  // outer 'a' has been hidden in this scope
19:         end
20: end
21: end test
```

After parsing the compound statement in function `func` (at line 11), you should output the symbol table with the following format.

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
d       variable    2(local)    real
-------------------------------------------------------
```

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

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
a       parameter   1(local)    integer
b       parameter   1(local)    real [2][3]
c       constant    1(local)    string       "hello world!"
-------------------------------------------------------
```

After parsing the compound statement in the main block (at line 19), you should output the symbol table with the following format.

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
a       variable    2(local)    boolean
-------------------------------------------------------
```

After parsing the main block (at line 20), you should output the symbol table with the following format.

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
a       variable    1(local)    integer
-------------------------------------------------------
```

After parsing the program's definition (at line 21), you should output the symbol table with the following format.

```
Name    Kind         Level      Type         Attribute
-------------------------------------------------------
test    program     0(global)   void
func    function    0(global)   boolean      integer, real [2][3]
-------------------------------------------------------
```

The output format of symbol table is as follows:

```
void dumpsymbol()
{
    int i;
    printf("%-32s\t%-11s\t%-11s\t%-17s\t%-11s\t\n","Name","Kind","Level","Type","Attribute");
    for(i=0;i< 110;i++)
        printf("-");
    printf("\n");
    {
        printf("%-32s\t", "func");
        printf("%-11s\t", "function");
        printf("%d%-10s\t", 0,"(global)");
        printf("%-17s\t", "boolean");
        printf("%-11s\t", "integer, real [2][3]");
        printf("\n");
    }
    for(i=0;i< 110;i++)
        printf("-");
    printf("\n");

}
```

Sample(s) will be later announced at the course forum.

4 How to Submit the Assignment?

You should create a directory, named "YourID" and store all files of the assignment under the directory. Once you are ready to submit your program, zip the directory as a single archive, name the archive as "YourID.zip", and upload it 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 reduced or, usually, **zero credit**.