Compiler Design

Description of

Project 2

Prof. Dr. Samy S. Abu Naser

Your are suppose to write program for the Syntax Analysis (the second Phase of the compiler design) using Java or any other Language.

We are going to implement Compiler for Pascal language (subset of the language).

To be able to do that you need

1- Grammar of the Pascal Language

2- A sample Pascal program to test with

1- Pascal Grammar

A.3 SYNTAX OF A PASCAL SUBSET

Listed below is an LALR(1) grammar for a subset of Pascal. The grammar can be modified for recursive-descent parsing by eliminating left recursion as described in Sections 2.4 and 4.3. An operator-precedence parser can be constructed for expressions by substituting out for **relop, addop,** and **mulu** and eliminating Î -productions. The addition of the production
statement -> **if** *expression* **then** *statement*

introduces the "dangling-else" ambiguity, which can be eliminated as discussed in Section 4.3 (see also Example 4.19 if predictive parsing is used).

There is no syntactic distinction between a simple variable and the call of function without parameters. Both are generated by the production
factor -> **id**

Thus, the assignment a := b sets a to the value returned by the function b, if b has been declared to be a function.

1- Pascal Grammar

program ->

program id ;

declarations

subprogram_declarations

compound_statement

.

identifier_list ->

id

| *identifier-list* , **id**

declarations ->

declarations **var** *identifier-list* : *type* ;

| є

type ->

standard_type

| **array [num . . num] of** *standard_type*

standard_type ->

integer

| **real**

subprogram_declarations ->

subprogram_declarations subprogram_declaration ;

| є

1- Pascal Grammar

subprogram_declaration ->
subprogram_head declarations compound_statement

subprogram_head ->
function id *arguments* : *standard_type* ;
| **procedure id** *arguments* ;
arguments ->
(*parameter-list*)
| ϵ

parameter_list ->
identifier_list : *type*
| *parameter_list* ; *identifier_list* : *type*

compound_statement ->
begin
optional_statements
end

optional_statements ->
statement_list
| ϵ

1- Pascal Grammar

statement_list ->
statement
| *statement_list* ; *statement*
statement ->
variable **assignop** *expression*
| *procedure_statement*
| *compound-statement*
| **if** *expression* **then** *statement* **else** *statement*
| **while** *expression* **do** *statement*

variable ->
id
| **id** [*expression*]
procedure_statement ->
id
| **id** (*expression-list*)

expression_list ->
expression
| *expression_list* , *expression*

expression ->
simple_expression
| *simple_expression* **relop** *simple_expression*

1- Pascal Grammar

simple_expression ->
term
| *sign term*
| *simple_expression* **sign** *term*
| *simple_expression* **or** *term*

term ->
factor
| *term* **mulop** *factor*

factor ->
variable
| **id** (*expression_list*)
| **num**
| (*expression*)
| **not** *factor*

2- A sample program to test with (p.txt)

```
program example;
var x, y: integer;
function gcd(a, b: integer): integer;
begin
if b = 0 then gcd := a
else gcd := gcd(b, a mod b)
end;

begin
read(x, y);
write(gcd(x, y))
end.
```

Implementation

1. You need to clean the grammar before you start from

- Left Factoring
- Left Recursion

2. Use phase 1 to get nextToken

3. The output of phase 2 is no errors or a list of errors

How to start the implementation

1) You need to have a method called match

// match job is the match token coming from the grammar with the
//token coming from phase one(lexical analyzer)
// it takes 2 arguments tn = token name. Tt = token type; it return true or false
// Sodo code for match

```
Function match(Tn: string, Tt :integer ): Boolean
    begin
       if ((tn==0) && (tn = token.name) && (tt = token.type)) | |
           ((tt in [1,2]) && (tt = token.type)) then
           nextToken()
       else begin
                  error(tn,tt);   // call error and print error message
                  system.exit(0);    // use this line if your want you compiler to
                                    // stop after the first error
                  nextToken();     // otherwise use this line to continue you compiler
                end;
End;
```

How to start the implementation

2) You need to have a method called error

// error job is to print an error message to the user about the error

// it takes 2 arguments Tn = token name. Tt = token type;

// Sodo code for error

```
Procedure error(Tn: string, Tt :integer )
  begin
    if ((tn==0)  then  writeln(“>>>  It is expected to have a keyword “,
                                           token.name, “ in line:”, token.lineno, “<<<“)

   else if ((tn==1)  then writeln(“>>>  It is expected to have a numeric constant in
                                          line:”, token.lineno, “<<<“)

   else if ((tn==2)  then writeln(“>>>  It is expected to have an identifier in line:”,
                                          token.lineno, “<<<“)
  end;
```

How to start the implementation

3) Here is a part of the grammar

```
program ->
    program id ;
   declarations
   subprogram_declarations
   compound_statement
   .
```

```
compound_statement ->
      begin
      optional_statements
      end
```

```
Procedure program;
  begin
     match(“program”, 0);
     match(“”, 2);
     match(“;”,0);
     declarations;
     subprogram_declarations;
     compound_statement ;
    match(“.”,0);
end;
```

```
Procedure compound_statement ;
  begin
     match(“begin”, 0);
     optional_statements;
     match(“end”);
 end;

Procedure optional_statements;
  begin
    -
    -
    -
  end;
```

How to start the implementation

3) Here is a part of the grammar -continue

```
program ->
   program id ;
  declarations
  subprogram_declarations
  compound_statement
  .
```

```
compound_statement ->
     begin
     optional_statements
     end
```

```
Procedure declarations;;
  begin
    -
    -
    -

end;
```

```
Procedure subprogram_declarations;
 begin
   -
   -
   -

 end;
```

When you finish your project submit it using the model :

Remember

- Add the names of project team