Marius Mitea
5508878

1. The Modified Grammar

I have changed the grammar a bit to get rid of the ambiguities and left recursions, to get rid of the * elements in the original grammar (they have been replaced by entirely new rules who's name is the original element with star added to it, as in `<classdecl>*` became `<classdeclstar>`, as well as for a couple of very particular reasons I'll explain below.

```
<prog>            ::= <classdeclstar> <progbody>
<classdeclstar>   ::= ε | <classdecl> <classdeclstar>
<classdecl>       ::= class id ( <definitionstar> ) ;                (1)
<progbody>        ::= program <funcbody> ; <definitionstar>          (1)
<definitionstar>  ::= ε | <definition> <definitionstar>
<definition>      ::= <type> id <definition2>
<definition2>     ::= ( <fparams> ) <funcbody> ;
                    | <arraystar> ;
<funcbody>        ::= { <typedeclstar> <statementstar> }
<typedeclstar>    ::= ε | <typedecl> <typedeclstar>
<typedecl>        ::= <type> id <arraystar> ;
<statementstar>   ::= ε | <statement> <statementstar>                (7)
<statement>       ::= <variable> := <expr> ;
                    | if ( <expr> ) then <statblock> else <statblock> fi ;
                    | for ( <statement> ; <expr> ; <statement> ) <statblock> ;
                    | cin ( <variable> ) ;
                    | cout ( <expr> ) ;
                    | return ( <expr> ) ;
<statblock>       ::= { <statementstar> } | <statement> | ε
<expr>            ::= <arithexpr> <expr2>                            (2)
<expr2>           ::= ε | relop <arithexpr>                          (3)
<arithexpr>       ::= <term> <arithexpr2> | <sign> <term> <arithexpr2>    (2)
<arithexpr2>      ::= ε | addop <term> <arithexpr2>                  (3)
<sign>            ::= + | -
<term>            ::= <factor> <term2>                               (2)
<term2>           ::= ε | multop <factor> <term2>                    (3)
<factor>          ::= <idnest> <factor2>                             (2,5)
                    | num
                    | ( <expr> )
                    | ! <factor>
<factor2>         ::= <indicestar>                                   (3)
                    | ( <aparams> )
<idnest>          ::= id <indicestar> <idnest2>                      (4)
<idnest2>         ::= ε | . <idnest>
<variable>        ::= <idnest> <indicestar>                          (5)
<indicestar>      ::= ε | <indice> <indicestar>
<indice>          ::= ε | [ <arithexpr> ]
<arraystar>       ::= ε | <array> <arraystar>
<array>           ::= ε | [ <int> ]
<type>            ::= int | float | id
<fparams>         ::= ε | <type> id <arraystar> <fparamstailstar>
<fparamstailstar> ::= ε | <fparamstail> <fparamstailstar>
<fparamstail>     ::= ε | , <type> id <arraystar>
<aparams>         ::= ε | <arithexpr> <aparamstailstar>
```

```
<aparamstailstar> ::= ε | <aparamstail> <aparamstailstar>
<aparamstail>     ::= ε | , <arithexpr>
<int>                                                            (6)
```

Now for the explanations of the more interesting changes:

(1) I merged the original `<typedecl>` and `<function>` into just one rule, `<definition>`, because both of them start the exact same, an identifier for the type followed by an identifier for the name and just now I can say whether its a function or a variable based on whether a semicolon (or [..];) or a (...) follows. As an added benefit, this way I can just use functions and variables in any order I want at absolutely no extra cost.

(2) These rules are either left recursions or ambiguous rules reduced to the base part and the problematic part split in (3).

(3) The problematic parts of (2).

(4) `<idnest>` had an issue where it would create an ambiguity in `<factor>` because both `<variable>` and `<idnest>*id(<aparams>)` start both with an id and it was impossible to distinguish between them. So I just rearranged `<idnest>` to first read the id and its [..], and then add a dot followed by `<idnest>` as needed. The result is exactly the same, except that I don't get the function name clearly anymore, so the semantics parser will be a bit harder to implement for this, only now I can group the 2 problematic parts into a common base in `<factor>` (5).

(5) The changes implied by (4).

(6) This is a special rule that parses a “number” as returned by the lexer and then verifies that it is an integer (no dot in it).

(7) The only other exception from how the normal rules are parsed, (7) forces the `<typedeclstar>` to break the cycle when the 2nd next token isn't an identifier. This is needed to be able to separate between `<typedeclstar>` and `<statementstar>` in functions and I couldn't just use the trick in (1) because I needed `<statementstar>` alone. To implement this I needed to add one more function to the lexer to return one token without “consuming” it.

2. The First and Follow Tables

Rule	FIRST	FOLLOW
`<prog>`	`class, program`	`$`
`<classdeclstar>`	`class, ε`	`program`
`<classdecl>`	`class`	`program`
`<progbody>`	`program`	`$`
`<definitionstar>`	`int, float, id, ε`	`), $`
`<definition>`	`int, float, id`	`), $`
`<definition2>`	`(, [, ε`	`), $`
`<funcbody>`	`{`	`;`
`<typedeclstar>`	`int, float, id, ε`	`id, if, for, cin, cout, return` **Note**: because of the ambiguity of having **`id`** in both the first and follow sets, the `<typedeclstar>` rule was manually fixed in code.
`<typedecl>`	`int, float, id`	`id, if, for, cin, cout, return`
`<statementstar>`	`id, if, for, cin, cout, return, ε`	`}`
`<statement>`	`id, if, for, cin, cout, return`	`)`
`<statblock>`	`{`	`else, fi, ;`
`<expr>`	`+, -, id, num, (, !`	`;, )`
`<expr2>`	`relop, ε`	`), ;`
`<arithexpr>`	`+, -, id, num, (, !`	`relop, addop, ], ,, ), ;`
`<arithexpr2>`	`addop, ε`	`relop, addop, ], ,, ), ;`
`<sign>`	`+, -`	`id, num, (, !`
`<term>`	`id, (, !, num`	`multop, ), ,, relop, addop, ;`
`<term2>`	`multop, ε`	`addop, ), ;, relop, ], ,, multop`
`<factor>`	`id, num, (, !`	`multop, relop, addop, ,, ), ;`
`<factor2>`	`id, [, ε`	`multop, addop, ,, ), relop, ;, (`

`<idnest>`	`id`	`[, (, ., ;, ), relop, addop, ], ,, :=, multop`
`<idnest2>`	`., ε`	`[, (, ., ;, ), relop, addop, ], ,, :=, multop`
`<variable>`	`id`	`:=, addop, multop, relop, ,, ), ;`
`<indicestar>`	`[, ε`	`., :=, ), multop, addop, ,, relop, ;, (`
`<indice>`	`[, ε`	`id, ., :=, ), addop, multop, ,`
`<arraystar>`	`[, ε`	`), ;, ,`
`<array>`	`[, ε`	`), ;, ,`
`<fparams>`	`int, float, id, ε`	`)`
`<fparamstailstar>`	`,, ε`	`)`
`<fparamstail>`	`,, ε`	`)`
`<aparams>`	`+, -, id, num, (, !`	`)`
`<aparamstailstar>`	`,, ε`	`)`
`<aparamstail>`	`,, ε`	`)`
`<int>`	`num`	`]`

3. The Design of the Parser

The parser is a text-book recursive descent predictive parser (literally), where I match every non-terminal with a corresponding function call. The functions' name starts with NTF_ for non-terminal function and the exact grammar rule name follows.

I made the Match(..) function accept either a string for the token text (only really used for convenience with reserved keywords since the lexical group is so vague) or a token type (used for everything else).

The constructor for the parser accepts a stream that it passes to the embedded lexer, and nothing else is kept as a state variable except the lookahead token and the output stream.

During the construction of the parser I needed to modify the lexer to:

- add the class keyword to the reserved keywords
- change || and && from the *logical* token type to the either *additive* token type or *multiplicative* token type according to the grammar
- include a function to just peek at the next token without actually consuming it, to get around a limitation of the grammar (I could have maybe avoided this)
- changed the function to return the next token to ignore comments

4. Usage and Examples

The compiler is made to be used like every other compiler out there, it's run with the executable name followed by the file it needs to compile.

Example command line: `jc test.jc`

An example test.jc file is attached next to showcase the compiler working fine. This is also attached to the archive containing this file.

```
class Foo
{
  int kind;
  float foo[4][3];

  int getKind()
  {
    return(kind[1]+2+3+4);
  };
};

program
{
  // variable declaration
  Foo foo;

  // complex expression
  cout(foo[0][1].var[0][1][2].getTick());//.getKind());

  // for with no body
  for(i:=0;;i<100;i:=i+1;);

  // for with 1 statement as body
  for(i:=0;;i<100;i:=i+1;) t:=meep();;

  // for with a block as body
  for(i:=0;;i<100;i:=i+1;)
  {
    t:=meep();
    if(meep())then
      cout(a);
    else
    {
      t:=!meep2();
      t:=t||!t;
    }
    fi;
  };

  cin(a);
  t:=meep;
};
```