COOL Parser description

Reyhaneh Derafshi
610396098

Introduction

This assignment is about constructing an AST using semantic actions of parser generator. Parser takes the sequence of tokens as input from the lexer and it produces a parse tree of the program and also handle some errors.
The root of the tree is of type program and the leaves are non-terminals. We are building our tree using context free grammar.

Overall layout

%{
C declarations
%{
Bison declarations
%%
Grammar rules
%%
Additional C code

- Bison declarations : The Bison declarations section contains declarations that define terminal and nonterminal symbols, specify precedence, and so on.

syntax

- **Precedence declarations** :

The declarations %left and %right ([left and] right associativity) take the place of %token which is used to declare a token type name without associativity/precedence.
e.i : %right ASSIGN

- **grammar rules** :

```
result: components...
        ;
```

exp: exp '+' exp

says that two groupings of type exp, with a `+' token in between, can be combined into a larger grouping of type exp.

Scattered among the components can be actions that determine the semantics of the rule. An action looks like this:

{C statements}

result: rule1-components…
| rule2-components…
…

The C code in an action can refer to the semantic values of the components matched by the rule with the construct $n, which stands for the value of the nth component. The semantic value for the grouping being constructed is $$

exp: …
| exp '+' exp
{ $$ = $1 + $3; }

code explanation

```
| CLASS TYPEID '{' error '}' ';' { yyclearin; $$ = NULL; }
| CLASS error '{' features_list '}' ';' { yyclearin; $$ = NULL; }
| CLASS error '{' error '}' ';' { yyclearin; $$ = NULL; }
```

handling errors in class definitions and body

```
features_list : features { $$ = $1; }
| { $$ = nil_Features();
features :
| error ';' { yyclearin; $$ = NULL; }
```

nonterminal features_list can be empty,
but we cannot allow the features nonterminal to call nil_Features()

formals are comma-separated arguments, i.e. "formal parameters"

```
formals :
| { $$ = nil_Formals(); }
```

empty argument list allowed

expressions are the body of the program

```
expr : OBJECTID ASSIGN expr { $$ = assign($1, $3); }

| '{' one_or_more_expr '}' { $$ = block($2); }
```

block of expression(s)

```
  | LET let_expr { $$ = $2; }
```

nested lets

```
    | CASE expr OF case_branch_list ESAC { $$ = typcase($2, $4); }
```

Use `case_branch_list` nonterminal to handle one or more cases

```
      one_or_more_expr   : expr ';' { $$ = single_Expressions($1); }
      | one_or_more_expr expr ';' { $$ = append_Expressions($1, single_Expressions($2));}
```

one or more expressions, separated by a semicolon
this is not the same as comma-separated expressions (e.g. a list of arguments)

```
 | error ';' { yyclearin; $$ = NULL; }
```

recover from an expression inside a block

```
param_expr : expr
                    | { $$ = nil_Expressions(); }
```

include nil because params are optional

```
      case_branch_list      : case_branch { $$ = single_Cases($1); }
```

must have at least one case_branch

output: