Inferencia de Tipos

La *inferencia de tipos* es la capacidad de deducir, ya sea parcial o totalmente, el tipo de una expresión en tiempo de compilación. El objetivo de este proyecto es la implementación de un intérprete de *COOL (Classroom Object-Oriented Language)* que posea inferencia de tipos mediante la adición del tipo **AUTO_TYPE**. Para más información consulte las especificaciones del proyecto en el documento Segundo proyecto de Compilacion.

Sobre los autores

Liset Silva Oropesa l.silva@estudiantes.matcom.uh.cu

Ariel Plasencia Díaz a.plasencia@estudiantes.matcom.uh.cu

Sobre el lenguaje COOL

Usted podrá encontrar la especificación formal del lenguaje **COOL** en el documento COOL Language Reference Manual.

Ejecutando la aplicación

Para lanzar la aplicación de escritorio, ejecute las siguientes instrucciones:

```
$ cd src/
$ make visual
```

Sin embargo puede correr el proyecto en consola mediante las líneas, donde aperece un fichero *.cl* predeterminado:

```
$ cd src/
$ make console
```

Además puede ejecutar el proyecto en consola mediante las líneas, pasando como primer y único parámetro fichero un fichero en *COOL*. *.cl* predeterminado:

```
$ cd src/
$ python3 type_inferer_console <fichero.cl>
```

Sobre la gramática

La gramática base con la cual se reconoce el lenguaje es la especificada en clases prácticas, la cual se muestra a continuación.

```
<program>                ::= <class_list>

<class_list>             ::= <def_class> <class_list>
                         |   <def_class>

<def_class>              ::= class TYPE { <feature_list> } ;
                         |   class TYPE inherits TYPE
{ <feature_list> } ;

<feature_list>           ::= <feature> <feature_list>
                         |   <empty>

<feature>                ::= ID : TYPE ;
                         |   ID : TYPE <- <expression> ;
                         |   ID ( <param_list> ) : TYPE
{ <expression> } ;
                         |   ID ( ) : TYPE { <expression> } ;

<param_list>             ::= <param>
                         |   <param> , <param_list>

<param>                  ::= ID : TYPE

<expression>             ::= if <expression> then <expression>
else <expression> fi
                         |   while <expression> loop
<expression> pool
                         |   { <expr_list> }
                         |   let <let_list> in <expression>
                         |   case <expression> of <case_list>
esac
                         |   ID <- <expression>
                         |   <truth_expr>

<expr_list>              ::= <expression> ;
                         |   <expression> ; <expr_list>

<let_list>               ::= ID : TYPE
                         |   ID : TYPE <- <expression>
                         |   ID : TYPE , <let_list>
                         |   ID : TYPE <- <expression> ,
<let_list>

<case_list>              ::= ID : TYPE => <expression> ;
```

```
                          |   ID : TYPE => <expression> ;
<case_list>

<truth_expr>              ::= not <truth_expr>
                          |   <comp_expr>

<comp_expr>               ::= <comp_expr> <= <arith>
                          |   <comp_expr> < <arith>
                          |   <comp_expr> = <arith>
                          |   <arith>

<arith>                   ::= <arith> + <term>
                          |   <arith> - <term>
                          |   <term>

<term>                    ::= <term> * <factor>
                          |   <term> / <factor>
                          |   <factor>

<factor>                  ::= isvoid + <factor_2>
                          |   <factor_2>

<factor_2>                ::= ~ <atom>
                          |   <atom>

<atom>                    ::= <atom> <func_call>
                          |   <member_call>
                          |   new TYPE
                          |   ( <expression> )
                          |   ID
                          |   INTEGER
                          |   STRING
                          |   TRUE
                          |   FALSE

<func_call>               ::= . ID ( <arg_list> )
                          |   . ID ( )
                          |   @ TYPE . ID ( <arg_list> )
                          |   @ TYPE . ID ( )

<arg_list>                ::= <expression>
                          |   expression , <arg_list>

<member_call>             ::= ID ( <arg_list> )
                          |   ID ( )
```

Sobre la implementación

Arquitectura

La arquitectura de la aplicación consiste primeramente en un lexer, el cual nos transforma una entrada de código *COOL* en una serie de tokens que lo representan y detecta errores respecto a la escritura incorrecta de símbolos. La segunda fase del compilador es el parser, el cual nos transforma la serie de tokens devueltos por el lexer en un *Árbol de Sintaxis Abstracta (AST)* que nos representa el código del programa. La siguiente y última fase es el análisis semántico en donde se verifica el cumplimiento de las reglas semánticas del lenguaje y la inferencia de tipos.

Análisis Lexicográfico

Las implementaciones del lexer fueron realizadas utilizando la biblioteca de python *ply*. La gramática base con la cual se reconoce el lenguaje es la especificada en las clases prácticas. En este punto se definieron los tokens de *COOL* mediante expresiones regulares, además de otras construcciones del lenguaje. Con esta herramienta se asegura que todos los tokens reconocidos sean los definidos en *COOL*.

Análisis Sintáctico

En esta segunda fase se utiliza la salida del lexer y la gramática atributada para lograr conformar el *Árbol de Sintaxis Abstracta* de *COOL* y a su vez verificar que la organización de los tokens reconocidos cumpla los requisitos del lenguaje. El parser escogido fue el *LR(1)*, siendo este uno de los más usados.

Análisis Semántico

Usando el patron visitor, se hacen cuatro recorridos por el *AST* con el objetivo de lograr el cumplimiento de las reglas de tipado y de inferencia de tipos presentes en *COOL*.

1. **Types Collector:** Se encarga de darle una pasada al código para coleccionar todos los tipos definidos en el programa.

2. **Types Builder:** Una vez conocidos todos los tipos involucrados en el programa en cuestión, esta estructura se encarga de formar la jerarquía de tipos.

3. **Types Checker:** Una vez conocida la jerarquía de tipos definida en el programa, esta estructura se encarga de realizar el chequeo de tipos de cada una de las expresiones definidas en el programa.

4. **Type Inferer:** Luego de realizar el chequeo de tipos, llevamos a cabo la inferencia de tipos a todos aquellos que se le asocia la palabra reservada *AUTO_TYPE*.

Sobre la aplicación

Esta aplicación puede ser ejecutada en los sistemas operativos *window* y *linux*, siempre y cuando estén instalados las librerías *ply*, *PyQt5* y *pydot* del lenguaje de programación *python*.