Cool Compiler Project

Integrantes

- Joel David Hernández Cruz
- Juan José Roque Cires
- Julio César Sánchez García

Content

Development Status.

Compiler Stage	Python Module	Status
Lexical Analysis	lexer.py	**complete**
Parsing	parser.py	**complete**
Semantic Analysis	semantic_analizaer.py	**complete**
Code Generation	mips_generator.py;	**complete**

Lexer & Parser

El proyecto esta desarrollado en python, para el desarrollo del lexer y el parser usamos la herramienta de parsing **ply**, que es una implementación de **Lex** y **Yac** originales de **C**, en python.

Grámatica

La Grámatica usada es libre de contexto y de recursión extrema izquierda, los problemas de ambiguedad que esto puede traer, se resuelven luego en el parser, definiendo ciertas reglas de presedencia para los tokens. Ejemplo:

```
precedence = (
     ('left', 'PLUS', 'MINUS'),
     ('left', 'TIMES', 'DIVIDE'),
 )

 # Esta declaracion define que PLUS/MINUS tienen el mismo nivel de presedencia y son
asociativos a la izquierda, al igual que TIMES/DIVIDE. Dentro de esta declaracion los
tokens son ordenados de baja a alta presedencia, luego esta declaracion definiria que
TIMES/DIVIDE tienen un mayor nivel de presedencia que PLUS/MINUS
```

```
<program>               ::= <classes>


<classes>               ::= <classes><class>;
```

```
                        |   <class>;

<class>                 ::= class TYPE <inherits> { <features_list_opt> }

<inhertits>             ::= inherits TYPE
                        |   <empty>

<features_list_opt>     ::= <features_list>
                        |   <empty>

<features_list>         ::= <features_list> <feature> ;
                        |   <feature> ;

<feature>               ::= ID ( <formal_params_list_opt> ) : TYPE { <expression> }
                        |   <formal>

<formal_params_list_opt>::= <formal_params_list>
                        |   <empty>

<formal_params_list>    ::= <formal_params_list> , <formal_param>
                        |   <formal_param>

<formal_param>          ::= ID : TYPE

<expr>                  ::= ID <- <expr>
                        |   <expr>.ID( <arguments_list_opt> )
                        |   <expr>@TYPE.ID( <arguments_list_opt> )
                        |   <if>
                        |   <while>
                        |   <block_expr>
                        |   <let>
                        |   <case>
                        |   new TYPE
                        |   isVoids <expr>
                        |   not <expr>
                        |   <expr> + <expr>
                        |   <expr> - <expr>
                        |   <expr> * <expr>
                        |   <expr> / <expr>
                        |   ~ <expr>
                        |   <expr> < <expr>
                        |   <expr> <= <expr>
                        |   <expr> = <expr>
                        |   <comment>
                        |   ( <expr> )
                        |   SELF
                        |   ID
                        |   STRING
                        |   TRUE
                        |   FALSE
                        |   INTEGER


<arguments_list_opt>      ::= <arguments_list>
```

```
                          |   <empty>

<arguments_list>        ::= <arguments_list_opt>, <expression>
                        |   <expression>

<case>                  ::= case <expression> of <case_actions> esac

<case_action>           ::= ID : TYPE => <expr>

<case_actions>          ::= <case_action>
                        |   <case_action> <case_actions>

<let_expression>        ::= let <formal_list> in <expression>

<formal_list>           ::= <formal_list>, <formal>
                        |   <formal>

<formal>                ::= ID : TYPE <- <expression>
                        |   ID : TYPE

<while>                 ::= while <expr> loop <expr> pool

<if>                    ::= if <expr> then <expr> else <expr> fi

<block_expr>            ::= { <block_list> }

<block_list>            ::= <block_list> <expr> ;
                        |   <expr> ;

<empty>                 ::=
```

Lexer

Para la implementación del lexer, definimos una lista de tokens, que define todos los posibles nombres que pueden tomar los tokens. Esta lista tambien es usada más adelante para identificar los terminales a la hora de parsear. Cada token esta especificado por una expresion regular compatible con el módulo **re** de Python. Ejemplo:

```
tokens = (
   'NUMBER',
   'PLUS',
)
# El nombre que le sigue a t_ debe machear exactamente con el del token correspondiente
t_PLUS = 'r\+'

# Para expresiones mas complejas la regla puede definirse como un metodo.
def t_NUMBER(t):
         r'\d+'
         t.value = int(t.value)
         return t
```

En el lexer definimos varios estados internos, para que en ciertos casos use diferentes reglas, tokens, etc.. Este es el caso para los **Strings** y los **Comentarios**. Para definir un nuevo estado del lexer primero debemos declararlo:

```
@property
    def states(self):
        return (
            ("STRING", "exclusive"),
            ("COMMENT","exclusive")
        )
```

Estos pueden tener dos tipos, **exclusive** e **inclusive**, en el 1ro se sobrescribe por completo el comportamiento del lexer, es decir el lexer solo aplicara reglas y retornara tokens definidos para este estado. El **inclusive** simplemente anhade reglas nuevas a las ya existentes. Luego para tokenizar strings y comentarios usamos los estados especiales definidos para ellos, y luego regresamos al estado por defecto.

Para el manejo de errores al tokenizar, ply ofrece varias facilidades para reportarlos.

```
def t_error(self, t):
        ...
        print("Illegal character! Line: {0}, character: {1}".format(t.lineno,
t.value[0]))
        ...
# Donde t.lineno seria el numero de la linea donde se encontro el error.
# Y t.valuep[0] el caracter que provoco este trigger.
```

Parser

Para la implementación del parser, hacemos uso de **ast.py**. Cada regla de la grámatica la definimos en Python, donde el docstring de la función contiene la especificación de la grámatica libre de contexto correspondiente. Ejemplo:

```
# Ejemplo de uso
def p_class(self, parse):
        """
        class : CLASS TYPE LBRACE features_list_opt RBRACE
        """
        parse[0] = AST.Class(name=parse[2], parent="Object", features=parse[4])
```

Como mecionamos anteriormente, los problemas de ambiguedad son resueltos por el parser al definir presedencia entre los operadores, el funcionamiento de esta, lo explicamos anteriormente con un ejemplo mas sencillo.

```
# precedence rules
    precedence = (
        ('right', 'ASSIGN'),
        ('right', 'NOT'),
        ('nonassoc', 'LTEQ', 'LT', 'EQ'),
        ('left', 'PLUS', 'MINUS'),
        ('left', 'MULTIPLY', 'DIVIDE'),
        ('right', 'ISVOID'),
        ('right', 'INT_COMP'),
        ('left', 'AT'),
        ('left', 'DOT')
    )
```

Para el manejo de errores al igual que en el lexer **ply.yacc** ofrece varias facilidades para reportarlos.

```
def p_error(self, parse):
        """
        Error rule for Syntax Errors handling and reporting.
        """
        if parse is None:
            error = "Error while trying to parse None... Unexpected end of input!"
            self.error_list.append(error)
        else:
            error = f"Syntax error! Line: {parse.lineno}, position: {parse.lexpos},
character: {parse.value}, type: {parse.type}"
            self.error_list.append(error)
            self.parser.errok()
# parse.line -> # de la linea
# parse.lexpos -> posicion relativa del caracter en el texto
# parse.value -> valor del caracter
# parse.type -> terminal
```

Ánalisis Semántico

Para el análisis semántico seguimos un patrón **visitor**. Todo el chequeo semántico se resuelve en una sola pasada. Realmente no todo, porque se realiza un precómputo para resolver los tipos disponibles, asi los métodos y atributos de cada uno pero realmente es O(N) siendo N la cantidad de tipos que se definen.

Luego es necesario recalcar observaciones que tuvimos, de las cuales algunas no se especifican el el manual de COOL.

- Si un método tiene como tipo de retorno **SELF_TYPE** su body tiene que retornar **SELF_TYPE** también. La razón para esto es para hahcer un compilador más Seguro para el Programador. Ya que puede generar problemas mas que visibles como:

```
class Animal{
    copy(): SELF_TYPE {new A};
};

class Perro : Animal{

};
```

Al intrntar `p : Perro <- (new Animal).copy()` esto es un error sematico ya que se intenta almacenar en una variable mas específico un objeto más general.

- Chequeamos errores de herencia cíclica asi como la imposibilidad de heredar de Int, Bool, String.
- Asociamos a cada nodo de AST su tipo estático que será de utilidad en tiempo de compilación.

Generación de Código

Cool -> Cil

Primero destacar que no nos manteniemos fieles al cien por ciento al CIL del Libro de Compilacion de Piad, basandonos en ese diseñamos un IL que pensamos que nos seria útil.

En la traducción a CIL resolvemos varios problemas.

- Renombramiento de variables y los label necesarios para cada salto en MIPS, esto ultimo fue debido a que el objetivo era que el tránsito CIL -> MIPS deberia ser lo mas sencillo posible.
- De la sencillez de CIL a MIPS, deriva que no tenemos un árbol CIL, si no una lista que se recorreria en orden y se transpilaria nodo a nodo, no todo nodo CIL tiene representacion escrita .
- Uno de los temas mas interesantes que nos encontramos aqui fue la resolución de case y un convenio para contruir los objetos, de esto ultimo hablaremos más abajo, para la resolución de los case primero se ordena cada action con el siguiente criterio, quien mas específico sea, para luego a travez de una funcion inherits saber si el tipo del expression del case hereda del tipo del action. Luego la pregunta seria como se define la funcion inherits. Sencillo con un DFS por los tipos disponibles marcamos para cada tipo el mometo de entrada en el DFS y el momento de salida, luego se puede afirmar que $T_1 < T_2$ si $llegada\ _{T_1} > llegada\ T_2$ y $salida\ _{T_1} < salida\ _{T_2}$.
- El otro punto, los contructores tratamos de diseñarlos como pensamos que lo haria c#, un metodo que se le agrega en esta face a cada clase, con nombre igual al de la clase, donde el body serian las asignaciones que se definen en los atributos. De esa manera crear un objeto se resuelve como un dispatch normal y no importa el orden en que se definan las clases, ni si una clase tiene como atributo un objeto de otra clase.
- Recopilamos todos los **String** que se definen en el código.
- Contruimos el .Type, donde en cada typo se guarda la información del dfs, el nombre del timpo para el type_name y las funciones que se definen para cada tipo con la invariante: Si $T_1 < T_2$ para todo método de T_2 si es el el i-esimo en su tabla virtual, también sera el i-esimo en la de T_1.

Cil -> Mips

La generación de mips se divide en 3 partes fundamentales:

1. Crear el constructor de la clase Main. Que sera el punto de entrada al programa.
2. **dotCode** donde se encuentran todo el código del programa
3. **dotData** donde se encuentran todas las definiciones de los strings

En Mips seguimos nuetro propio convenio, y trabajamos casi todo en pila. El siguiente convenio fue estudiado en **Programacion de Maquina**, cada llamado a funcion(un **dispatch**), pusheamos en la pila el valor de los registros `$fp` y `$ra` y estos son los unicos registros que salvamos, luego movemos el valor de `$sp` a `$fp` para simular una pila vacia, y al regresar del llamado realizamos la operacion contraria.

Usamos el registro `$fp` para acceder a los argumentos y a los locales en un metodo, a la derecha de `$fp` estan los argumentos y a la izquierda las locales.

A los tipos basicos [Int, String, Boolean] se les hace boxing y unboxing. Aclarar que al ser pasados como argumento se les hace una copia del valor no del tipo.

La tabla de metodos para cada tipo, se lleva en mips, donde a cada tipo se le asigna una direccion estatica, y a partir de ahi se colocan punteros a los metodos que estas definen.

Ejecutando el proyecto

En el proyecto incluimos un virtual enviorment, por lo que no es necesario instalar ninguna dependecia para correrlo, en **coolc.sh** activamos el enviorment y corremos el proyecto con los casos de pruebas especificados.

```
$ ./coolc.sh <input_file.cl>
```