Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Organización de Lenguajes y Compiladores 2

Guillermo Alfredo Peitzner Estrada – 201504468.

Manual Técnico

Lenguaje utilizado

Python 3.8: https://www.python.org/.

Librerías

PyQt5 5.15.0: https://pypi.org/project/PyQt5/.

ply 3.11: https://pypi.org/project/ply/.

graphviz 0.14: https://pypi.org/project/graphviz/.

Dependencias de terceros

Graphviz: https://graphviz.org/.

Licencia del software

GPL-3.0: http://www.gnu.org/licenses/gpl-3.0.html

Repositorio oficial

https://github.com/gpeitzner/OLC2_Proyecto2

Glosario de Términos

Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

PLY

PLY es una herramienta de análisis escrita exclusivamente en Python. Es, en esencia, una reimplementación de Lex y Yacc originalmente en lenguaje C. Fue escrito por David M. Beazley. PLY utiliza la misma técnica de análisis LALR que Lex y Yacc.

PyQT5

PyQt es un binding de la biblioteca gráfica Qt para el lenguaje de programación Python. La biblioteca está desarrollada por la firma británica Riverbank Computing y está disponible para Windows, GNU/Linux y Mac OS X bajo diferentes licencias.

Graphviz

Graphviz es un conjunto de herramientas de software para el diseño de diagramas definido en el lenguaje descriptivo DOT. Fue desarrollado por AT&T Labs y liberado como software libre con licencie tipo Eclipse.

Interprete

En ciencias de la computación, intérprete o interpretador es un programa informático capaz de analizar y ejecutar otros programas. Los intérpretes se diferencian de los compiladores o de los ensambladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción.

Patrón interprete

Es un patrón de diseño que, dado un lenguaje, define una representación para su gramática junto con un intérprete del lenguaje. Se usa para definir un lenguaje para representar expresiones regulares que representen cadenas a buscar dentro de otras cadenas.

Fuente del glosario de términos: https://es.wikipedia.org/.

Archivos de código fuente utilizados

Nombre	Descripción
AMBITO.PY	Contiene a la clase “Ámbito” y es utilizada en el intérprete de Augus.
ASCENDENTE.PY	Contiene el analizador léxico y sintáctico de MinorC, también se genera el árbol sintáctico.
ASCENDENTE_AUGUS.PY	Contiene el analizador léxico y sintáctico de Augus, también se genera el árbol sintáctico.
CLASES.PY	Contiene todas las clases utilizadas en la construcción del árbol sintáctico, sirviendo como modelo de los objetos que lo conforman.
EXPRESIONES.PY	Contiene todas las clases que representan expresiones y que son utilizadas para construir el árbol sintáctico de Augus.
INSTRUCCIONES.PY	Contienen todas las clases que representan una instrucción y que son utilizadas para construir el árbol sintáctico de Augus.
INTERPRETE.PY	Ejecuta el árbol sintáctico generado por los analizadores de Augus.
PARSERTAB.PY	Archivo generado por PLY.
PRINCIPAL.PY	Contiene toda la interfaz gráfica del programa, además que ejecuta las principales funciones solicitadas por el usuario final.
RECURSOS.PY	Contiene recursos gráficos utilizados en la interfaz gráfica.
TABLASIMBOLOS.PY	Contiene la tabla de símbolos utilizada por el intérprete del Augus.
TRES_DIRECCIONES.PY	Contiene todas las funciones necesarias para la generación de código de tres direcciones, optimizaciones y estructuras con contenido para la generación de reportes.
VALORES.PY	Contiene todas las clases utilizadas en la creación del árbol sintáctico de Augus.

Clases utilizadas en la creación del AST

Clase	Descripción
Declaracion	Clase que representa de manera general a una declaración.
DeclaracionFinal	Clase que representa específica a una declaración.
Estructura	Clase que representa a un “struct”.
Función	Clase que representa a una función.
Parametro	Clase que representa a un parámetro en una función.
Metódo	Clase que representa la llamada a un método.
Etiqueta	Clase que representa a una etiqueta.
Salto	Clase que representa a un salto.
AsignacionNormal	Clase que representa a una asignación estándar.
AsignacionEstructura	Clase que representa a una asignación en una estructura.
AsignacionAumento	Clase que representa una asignación con aumento.
AsignacionDecremento	Clase que representa una asignación con decremento.
_If	Clase que representa a la sentencia de control “IF”.
_ElseIf	Clase que representa a un “else if” de un “IF”.
_Else	Clase que representa al “else” de un “IF”.
_Switch	Clase que representa a la sentencia de control “Switch”.
_Case	Clase que representa a un “case” de un “Switch”.
_DefaultCase	Clase que representa a un “default” de un “Switch”.
_While	Clase que representa a una sentencia de control “While”.
_Do	Clase que representa a una sentencia de control “Do”.
_For	Clase que representa a una sentencia de control “For”.
Expresión	Clase que representa a una expresión aritmética, relacional, lógica, binaria, etc.
_SizeOf	Clase que representa a la función “sizeof()”.
Valor	Clase que representa a un entero, carácter, decimal o una cadena.
_Continue	Clase que representa a la función “continue”.
_Break	Clase que representa a la función “break”.
_Return	Clase que representa a la función “return”.

Generalidades

Construcción del árbol sintáctico abstracto

El árbol sintáctico abstracto fue construido utilizando clases que representaran de manera especifica a cada hoja, de esta forma se garantiza la realización sencilla de acciones complejas con la manipulación de hojas, esto se logra sintetizando los objetos y con la ayuda de listas.

Uso del patrón de interprete

El patrón interprete es usado en el momento de generar código de tres direcciones, ya que se recorre cada elemento del árbol en preorden y dependiendo de qué tipo de instancia sea la hoja, se realiza la generación de código de tres direcciones, para el manejo de ámbitos se utilizaron tres clases: Símbolo, TablaSimbolos y Ámbito, con estas clases se manejo el correcto uso de variables y su asociación con los temporales para la fácil generación de código.

Reporte gramatical

El reporte gramatical es generado cuando se realiza el análisis ascendente aprovechando que se realiza esta tarea para ir concatenando cada elemento de la tabla que conforma el reporte gramatical, esto mejora el resultado final de rendimiento de la aplicación.

Reporte de AST

Para poder generar el reporte del árbol sintáctico abstracto, se recorre todo el árbol y aprovechando de que cada hoja es una instancia de una clase, se comprueba y se concatena el código necesario utilizado por graphviz, al final este se muestra automáticamente haciendo uso de la biblioteca única para Python.

Optimización de código

Para la optimización de código se crearon funciones especificas para cada grupo de reglas, gracias al uso de expresiones regulares se pueden comprobar cada uno de los elementos para hacer validaciones por cada regla y se optimiza el código de manera incremental.

Principales funciones utilizadas en la generación de Código 3D

Método	Descripción
generar_codigo	Método de inicio de generación de código.
obtener_funciones	Carga las funciones en memoria.
mostrar_mensaje_consola	Muestra un mensaje en la terminal del editor.
existe_main	Comprueba que exista la función main en las funciones.
cargar_variables_globales	Carga las variables globales en memoria.
obtener_expresion	Genera el código de una expresión y devuelve el temporal de esta.
obtener_expresion_aritmetica	Genera el código de una expresión aritmética.
obtener_expresion_relacional	Genera el código de una expresión relacional.
obtener_expresion_logica	Genera el código de una expresión lógica.
obtener_expresion_bit	Genera el código de una expresión bit.
obtener_expresion_unaria	Genera el código de una expresión unaria.
obtener_expresion_ternaria	Genera el código de una expresión ternaria.
obtener_expresion_casteo	Genera el código de una expresión de casteo.
obtener_expresion_identificador_arreglo	Genera el código de un acceso a un arreglo.
obtener_expresion_estructura	Genera el código de un acceso a una estructura.
obtener_expresion_aumento_decremento	Genera el código de un acceso a una variable con aumento o decremento.
obtener_expresion_sizeof	Genera el código de un sizeof.
obtener_registro_temporal	Genera un registro temporal.
obtener_etiqueta_temporal	Genera una etiqueta temporal.
generar_codigo_main	Genera el código del método principal.
generar_codigo_instrucciones	Genera el código de instrucciones.
generar_codigo_etiqueta	Genera el código de una etiqueta.
generar_codigo_salto	Genera el código de un salto.
generar_codigo_declarcion	Genera el código de una declaración.
generar_codigo_asignacion	Genera el código de una asignación.
generar_codigo_if	Genera el código de un IF.
generar_codigo_switch	Genera el código de un SWITCH.
generar_codigo_break	Genera el código de un BREAK.
generar_codigo_while	Genera el código de un WHILE.
generar_codigo_do	Genera el código de un DO.
generar_codigo_for	Genera el código de un FOR.
generar_codigo_metodo	Genera el código de una llamada a un método.
generar_codigo_printf	Genera el código de un PRINTF.
optimizar_codigo	Optimiza el código generado.

Gramática Ascendente

PRODUCCION	REGLA
INIT -> CUERPO_GLOBAL	INIT.VAL = CUERPO_GLOBAL.VAL;
CUERPO_GLOBAL -> LISTA_GLOBAL	CUERPO_GLOBAL.VAL = LISTA_GLOBAL.VAL;
CUERPO_GLOBAL ->	CUERPO_GLOBAL.VAL = NONE;
LISTA_GLOBAL -> LISTA_GLOBAL1 INSTRUCCION_GLOBAL	LISTA_GLOBAL1.ADD(INSTRUCCION_GLOBAL); LISTA_GLOBAL.VAL = LISTA_GLOBAL1.VAL;
LISTA_GLOBAL -> INSTRUCCION_GLOBAL	LISTA_GLOBAL.VAL = Lista(INSTRUCCION_GLOBAL.VAL);
INSTRUCCION_GLOBAL -> DECLARACION; | ESTRUCTURA ; | FUNCION	INSTRUCCION_GLOBAL.VAL = DECLARACION.VAL | ESTRUCTURA.VAL | FUNCION.VAL;
ESTRUCTURA -> struct identificador { CARACTERISTICAS }	ESTRUCTURA.VAL = Estructura(identificador, CARACTERISTICAS.VAL, lineno);
CARACTERISTICAS -> LISTA_CARACTERISTICAS	CARACTERTISCIAS.VAL = LISTA_CARACTERISTICAS.VAL;
CARACTERISTICAS ->	CARACTERTISICAS.VAL = NONE;
LISTA_CARACTERISTICAS -> LISTA_CARACTERISTICAS CARACTERISTICA	LISTA_CARACTERISTICAS1.APPEND(CARACTERISTICA.VAL); LISTA_CARACTERISTICAS.VAL = LISTA_CARACTERISTICAS1.VAL;
LISTA_CARACTERITISCAS -> CARACTERISTICA	LISTA_CARACTERISTICAS.VAL = Lista(CARACTERISTICA.VAL);
CARACTERISTICA -> DECLARACION;	CARACTERISTICA.VAL = DECLARACION.VAL;
FUNCION -> TIPO identificador (PARAMETROS) { CUERPO_LOCAL }	FUNCION.VAL = Funcion(TIPO.VAL, identificador, PARAMETROS.VAL, CUERPO.VAL);
PARAMETROS -> LISTA_PARAMETROS	PARAMETROS.VAL = LISTA_PARAMETROS.VAL;
PARAMETROS ->	PARAMETROS.VAL = NONE;
LISTA_PARAMETROS -> LISTA_PARAMETROS1 , PARAMETRO	LISTA_PARAMETROS1.ADD(PARAMETRO.VAL); LISTA_PARAMETROS.VAL = LISTA_PARAMETROS1.VAL;
LISTA_PARAMETROS -> PARAMETRO	LISTA_PARAMETROS.VAL = Lista(PARAMETRO.VAL);
PARAMETRO -> TIPO identificador	PARAMETRO.VAL = Parametro(TIPO.VAL, false, identificador);
PARAMETRO -> TIPO & identificador	PARAMETRO.VAL = Parametro(TIPO.VAL, true, identificador);
CUERPO_LOCAL -> LISTA_LOCAL	CUERPO_LOCAL.VAL = LISTA_LOCAL.VAL;
CUERPO_LOCAL ->	CUERPO_LOCAL.VAL = NONE;
LISTA_LOCAL -> LISTA_LOCAL1 INSTRUCCION_LOCAL	LISTA_LOCAL1.ADD(INSTRUCCION_LOCAL.VAL); LISTA_LOCAL.VAL = LISTA_LOCAL1.VAL;
LISTA_LOCAL -> INSTRUCCION_LOCAL	LISTA_LOCAL.VAL = Lista(INSTRUCCION_LOCAL.VAL);

INSTRUCCION_LOCAL -> ETIQUETA | SALTO | DECLARACION ; | ASIGNACION ; | IF | SWITCH | WHILE | DO | FOR | PRINT ; | METODO ;	INSTRUCCION_LOCAL.VAL = ETIQUETA.VAL | SALTO.VAL | DECLARACION.VAL | ASIGNACION.VAL | IF.VAL | SWITCH.VAL | WHILE.VAL | DO.VAL | FOR.VAL | PRINT.VAL | METODO.VAL;
PRINT -> printf (LISTA_EXPRESIONES)	PRINT.VAL = PrintF(LISTA_EXPRESIONES.VAL, lineno);
INSTRUCCION_LOCAL -> continue;	INSTRUCCION_LOCAL.VAL = Continue(lineno);
INSTRUCCION_LOCAL -> break;	INSTRUCCION_LOCAL.VAL = Break(lineno);
INSTRUCCION_LOCAL -> return EXPRESION;	INSTRUCCION_LOCAL.VAL = Return(EXPRESION.val, lineno);
INSTRUCCION_LOCAL -> return;	INSTRUCCION_LOCAL.VAL = Return(NONE, lineno);
METODO -> identificador (EXPRESIONES)	METODO.VAL = Metodo(identidicador, EXPRESIONES.VAL, lineno);
ETIQUETA -> identificador:	ETIQUETA.VAL = Etiqueta(identificador, lineno);
SALTO -> goto identificador;	SALTO.VAL = Salto(identificador, lineno);
DECLARACION -> TIPO LISTA_DECLARACION	DECLARACION.VAL = Declaracion(TIPO.VAL, LISTA_DECLARACION.VAL);
LISTA_DECLARACION -> LISTA_DECLARACION1 , DECLARACION_FINAL	LISTA_DECLARACION1.ADD(DECLARACION_FINAL.VAL); LISTA_DECLARACION.VAL = LISTA_DECLARACION1.VAL;
LISTA_DECLARACION -> DECLARACION_FINAL	LISTA_DECLARACION.VAL = Lista(DECLARACION_FINAL.VAL);
DECLARACION_FINAL -> identificador INDICES	DECLARACION_FINAL.VAL = DeclaracionFinal(identificador, INDICES.VAL, NONE, lineno);
DECLARACION_FINAL -> identificador INDICES = EXPRESION	DECLARACION_FINAL.VAL = DeclaracionFinal(identificador, INDICES.VAL, EXPRESIONES.VAL, lineno);
INDICES -> ACCESOS	INDICES.VAL = ACCESOS.VAL
INDICES ->	INDICES.VAL = NONE;
ACCESOS -> ACCESOS1 ACCESO	ACCESOS1.ADD(ACCESO.VAL); ACCESOS.VAL = ACCESOS1.VAL;
ACCESOS -> ACCESO	ACCESOS.VAL = Lista(ACCESO.VAL);
ACCESO -> [EXPRESION]	ACCESO.VAL = EXPRESION.VAL;
ACCESO -> [ ]	ACCESO.VAL = Lista();
ASIGNACION -> identificador INDICES COMPUESTO EXPRESION	ASIGNACION.VAL = AsignacionNormal(identificador, INDICES.VAL, COMPUESTO.VAL, EXPRESION.VAL, lineno);
ASIGNACION -> identificador INDICES . identificador INDICES COMPUESTO EXPRESION	ASIGNACION.VAL = AsignacionEstructura(identificador, INDICES.VAL, identificador, INDICES.val, COMPUESTO.VAL, EXPRESION.VAL, lineno);
ASIGNACION -> identificador ++	ASIGNACION.VAL = AsignacionAumento(identificador, lineno);
ASIGNACION -> ++ identificador	ASIGNACION.VAL = AsignacionAumento(identificador, lineno);
ASIGNACION -> identificador --	ASIGNACION.VAL = AsignacionDecremento(identificador, lineno);

ASIGNACION -> -- identificador	ASIGNACION.VAL = AsignacionDecremento(identificador, lineno);
COMPUESTO -> = | += | -= | *= | /= | %= | <<= | >>= | & | |= | ^=	COMPUESTO.VAL = = | += | -= | *= | /= | %= | <<= | >>= | & | |= | ^=;
IF -> if (EXPRESION) { CUERPO_LOCAL }	IF.VAL = If(EXPRESION.VAL, CUERPO_LOCAL.VAL, NONE, NONE, lineno);
IF -> if (EXPRESION) { CUERPO_LOCAL } ELSE	IF.VAL = If(EXPRESION.VAL, CUERPO_LOCAL.VAL, NONE, ELSE.VAL, lineno);
IF -> if (EXPRESION) { CUERPO_LOCAL } ELSEIF IF_FINAL	IF.VAL = If(EXPRESION.VAL, CUERPO_LOCAL.VAL, ELSEIF.VAL, IF_FINAL.VAL, lineno);
IF_FINAL -> ELSEIF IF_FINAL	IF_FINAL.VAL = Lista(ELSEIF) + IF_FINAL.VAL;
IF_FINAL -> ELSE	IF_FINAL.VAL = Lista(ELSE.VAL);
IF_FINAL ->	IF_FINAL.VAL = NONE;
ELSEIF -> else if (EXPRESION) { CUERPO_LOCAL }	ELSEIF.VAL -> ElseIf(EXPRESION.VAL, CUERPO_LOCAL.VAL);
ELSE -> else { CUERPO_LOCAL }	ELSE.VAL = Else(CUERPO_LOCAL.VAL);
SWITCH -> switch (EXPRESION) { CASES DEFAULT_CASE }	SWITCH.VAL = Switch(EXPRESION.VAL, CASES.VAL, DEFAULT_CASE.VAL, lineno);
CASES -> LISTA_CASE	CASES.VAL = LISTA_CASE.VAL
CASES ->	CASES.VAL = NONE;
LISTA_CASE = LISTA_CASE1 CASE	LISTA_CASE1.ADD(CASE.VAL); LISTA_CASE.VAL = LISTA_CASE1.VAL;
LISTA_CASE -> CASE	LISTA_CASE.VAL = Lista(CASE.VAL);
CASE -> case EXPRESION : CUERPO_LOCAL	CASE.VAL = Case(EXPRESION.VAL, CUERPO_LOCAL.VAL);
DEFAULT_CASE -> default : CUERPO_LOCAL	DEFAULT_CASE.VAL = DefaultCase(CUERPO_LOCAL.VAL);
WHILE -> while (EXPRESION) { CUERPO_LOCAL }	WHILE.VAL = While(EXPRESION.VAL, CUERPO_LOCAL.val, lineno);
DO -> do { CUERPO_LOCAL } while (EXPRESION) ;	DO.VAL = Do(CUERPO_LOCAL.VAL, EXPRESION.VAL, lineno);
FOR -> for (INICIO_FOR ; EXPRESION ; ASIGNACION) { CUERPO_LOCAL }	FOR.VAL = For(INICIO_FOR.VAL, EXPRESION.VAL; ASIGNACION.VAL, CUERPO_LOCAL.VAL, lineno);
INICIO_FOR -> DECLARACION | ASIGNACION	INICIO_FOR.VAL = DECLARACION.VAL | ASIGNACION.VAL;
EXPRESION -> EXPRESION1 [+|-|*|/|%] EXPRESION2	EXPRESION.VAL -> ExpresionAritmetica(EXPRESION1.VAL, [+|-|*|/|%], EXPRESION2.VAL);
EXPRESION -> EXPRESION1 [==|!=|>|<|>=|<=] EXPRESION2	EXPRESION.VAL = ExpresionRelacional(EXPRESION1.VAL, [==|!=|>|<|>=|<=], EXPRESION2.VAL);
EXPRESION -> EXPRESION1 [and|or] EXPRESION2	EXPRESION.VAL = ExpresionLogica(EXPRESION1.VAL, [and|or], EXPRESION2.VAL);
EXPRESION -> EXPRESION1 [<<|>>|&|"|"|^] EXPRESION2	EXPRESION.VAL = ExpresionBit(EXPRESION1, [<<|>>|&|"|"|^], EXPRESION2);
EXPRESION -> EXPRESION1 ? EXPRESION2 : EXPRESION3	EXPRESION.VAL = ExpresionTernaria(EXPRESION1, EXPRESION2, EXPRESION3);
EXPRESION -> [-|!|~] EXPRESION1	EXPRESION.VAL = ExpresionUnaria([-|!|~], EXPRESION1);

EXPRESION -> & identificador	EXPRESION.VAL = ExpresionReferencia(identificador);
EXPRESION -> METODO	EXPRESION.VAL = METODO.VAL;
EXPRESION -> (EXPRESION1)	EXPRESION.VAL = EXPRESION1.VAL;
EXPRESION -> identificador INDICES . identificador INDICES	EXPRESION.VAL = ExpresionEstructura(identificador, INDICES.VAL, identificador, INDICES.VAL);
EXPRESION -> identificador ACCESOS	EXPRESION.VAL = ExpresionIdentificadorArreglo(identificador, ACCESOS);
EXPRESION -> { EXPRESIONES }	EXPRESION.VAL = ExpresionElementos(EXPRESIONES.VAL);
EXPRESION -> sizeof (TIPO)	EXPRESION.VAL = SizeOf(TIPO.VAL);
EXPRESION -> identificador ++	EXPRESION.VAL = ExpresionAumentoDecremento(identificador, ++, post);
EXPRESION -> ++ identificador	EXPRESION.VAL = ExpresionAumentoDecremento(identificador, ++, pre);
EXPRESION -> identificador --	EXPRESION.VAL = ExpresionAumentoDecremento(identificador, --, post);
EXPRESION -> -- identificador	EXPRESION.VAL = ExpresionAumentoDecremento(identificador, --, pre);
EXPRESION -> scanf ( )	EXPRESION.VAL = ExpresionScan();
EXPRESION -> (TIPO) EXPRESION	EXPRESION.VAL = ExpresionCasteo(TIPO.VAL, EXPRESION.VAL);
EXPRESION -> carácter	EXPRESION.VAL = caracter;
EXPRESION -> cadena	EXPRESION.VAL = cadena;
EXPRESION -> entero	EXPRESION.VAL = entero;
EXPRESION -> decimal	EXPRESION.VAL = decimal;
EXPRESION -> identificador	EXPRESION.VAL = identificador;
EXPRESIONES -> LISTA_EXPRESIONES	EXPRESIONES.VAL = LISTA_EXPRESIONES.VAL
EXPRESIONES ->	EXPRESIONES.VAL = NONE;
LISTA_EXPRESIONES -> LISTA_EXPRESIONES1 , EXPRESION	LISTA_EXPRESIONES1.ADD(EXPRESION.VAL); LISTA_EXPRESIONES.VAL = LISTA_EXPRESIONES1.VAL;
LISTA_EXPRESIONES -> EXPRESION	LISTA_EXPRESIONES.VAL = Lista(EXPRESION.VAL);
TIPO -> int | char | double | float | void	TIPO.VAL = Tipo([int|char|double|float|void], None);
TIPO -> struct identificador	TIPO.VAL = Tipo(struct, identificador);