Instituto Tecnológico de Costa Rica
Escuela de Computación
Ingeniería en Computación
Sede Central

IC-5701 Compiladores e Intérpretes

Proyecto III: Generador de Código

Integrantes:

José Daniel Camacho
carné:2017043395
Andrey Torres Calderón
carné: 2017079723
Eduardo Jirón Alvarado
carné: 2017101878
Kristal Durán Araya
carné: 2017238958

Profesor: Ignacio Trejos Zelaya

I Semestre,
2019

Tabla de contenidos

Introducción

El siguiente informe es una documentación externa del tercer proyecto del curso IC-5701 Compiladores e Intérprete. En este documento se explican las soluciones a los principales problemas atacados, también enumera una lista de aspectos importantes como: los errores, las pruebas positivas y negativas realizadas agregando los resultados obtenidos a cada una de ellas, y las tareas realizadas divididas según la elaboración por miembro del equipo de trabajo. Se indican además las conclusiones y experiencias al finalizar este proyecto.

Este proyecto tiene el objetivo de retar a los estudiantes para conseguir extender el generador de código que posee Triángulo agregando retos en cuanto a algunos ciclos, manejo de recursividad, privacidad de variables, entre otros. Al final de este documento se encuentra un enlace para acceder al enunciado original y la última versión proporcionada por el profesor Ignacio Trejos.

Este proyecto se desarrolla como una tercer fase tomando como base el proyecto uno: análisis sintáctico y el proyecto dos: Analizador Contextual, desarrollados por los mismo estudiantes: Jose Daniel Camacho, Kristal Durán Araya, Eduardo Jiron y Andrey Torres. Ya que la fase dos tenía problemas para la ejecución satisfactoria de la instrucción recursive se realizan los cambios necesarios para completar esta etapa y así finalizar con la fase tres de la generación de código.

Desarrollo

Este apartado pretende mostrar una descripción general a la forma de abordar las soluciones de los principales problemas propuestos. Para ello se lista una serie de preguntas proporcionadas por el profesor Ignacio Trejos. También se muestra un listado de errores detectados, y un cuadro para mostrar las pruebas positivas y negativas que se realizaron.

Preguntas:

- Descripción del esquema de generación de código para el comando pass

  Para la generación de código del comando pass únicamente se retorna un null en las instrucciones EmptyCommand y EmptyExpression, por lo que no se agrega ningún dato a la pila, sin afectar la memoria, entradas o salidas, lo que genera un comportamiento nulo de su ejecución.

- Descripción del esquema de generación de código para el comando if_then_else_end.

  En la implementación del if no se realizan modificación adiciones, se deja la ejecución igual que para triángulo. El if funciona realizando inicialmente un

visit de la expresión en la cual se cargan los datos y se hace un llamado al operador correspondiente (<,>), posteriormente se realiza el JumpIf brincando a la opción del else (es decir se valida un false en la expresión) ejecutando así el comando 2, en caso de que la expresión sea true se sigue con la ejecución del comando 1, luego de cualquier de los dos llamados se finaliza la ejecución.

- Descripción del esquema de generación de código para el comando choose .

  Se crearon dos objetos CaseTree y CaseJumpsPatcher los cuales están en la misma carpeta del Encoder, CaseTree contiene una referencia a la expresión del choose, la cual permite hacer un load a la tabla cuando se necesite, ya que el CaseTree se pasa por el segundo parámetro de los visits de tipo Objeto, también contiene el frame y contiene el objeto CaseJumpsPatcher que funciona como una lista enlazada, la cual en cada nodo contiene las direcciones a los jumps que se generan para las comparaciones, el esquema que se utiliza es que primero se genera el código para el comando de un case, luego se generan las comparaciones las cuales se van agregando a la lista enlazada lo que permite tener todas las direcciones a las comparaciones y al final, de la última comparación se genera un jumpif al comando, si no se genera un jump al final del código o el siguiente caso.

  (**Nota 1**: las comparaciones si es un rango primero compara la expresión con el menor del rango y hacer un call a ge despues de eso tiene un jumpif false hacia el siguiente intervalo del case, en caso de no existir salta a ejecutar el caso por default o terminar la ejecución, luego del jumpif compara la expresión con el mayor del rango y hace un call luego tiene un jumpif true hacia el código del comando que a su vez tiene un jump al final del código, todas las direcciones a esos jumps se guardan en el CaseJumpsPatcher para luego ser parcheadas porque se necesita generan la mayoría del código primero para luego cambiar las direcciones de los jumps).

  (**Nota 2**: El árbol se recorre de atrás hacia adelante al igual que los rangos, esto sólo afecta al orden de la generación de código pero no al producto :D).

  (**Nota 3**: Si da error con algun caracter, probar cerrar el archivo, y volver a ejecutar, preferiblemente poder hacer clean and build antes de volver a ejecutar)

- Descripción del esquema de generación de código para todas las variantes de loop … end.
  - loop while do: primero se emite un salto hacia la evaluación de la expresión del while luego se emite un salto hacia el comienzo del comando del while si la expresión es verdadera
  - loop until do: primero se emite un salto hacia la evaluación de la expresión del while luego se emite un salto hacia el comienzo del comando del until si la expresión es falsa
  - loop do while: ejecuta el comando primero y luego evalúa la expresión del while, luego emite un jump si la expresión da como resultado true

- loop do until: ejecuta el comando primero y luego evalúa la expresión del until, luego emite un jump si la expresión da como resultado false

- Descripción del esquema de generación de código para loop for … end:

  Lo primero que hace es evaluar la segunda expresión y luego la primera, de esta manera la primera expresión queda en el top de la pila, luego hace un jump hacia la evaluación de la expresión 1 y 2 verifica que la 1 sea menor o igual a la 2, si es true continúa con la ejecución del comando del for y luego incrementa la expresión 1 que se encuentra en el top de la pila por lo que solo es necesario llamar a succ sin tener que cargar primero el valor y luego guardarlo, luego se repite el ciclo evaluando la expresión 1 y 2.

- Descripción del esquema de generación de código para loop for … end con while o con until:

  Es igual que el for solo que luego de evaluar la expresión 1 y 2 compara con con la expresión 3 haciendo un and y si resulta true continúa con la ejecución del comando, en el caso del until luego de evaluar la expresión 3 la niega para proceder con el and.
- Solución dada al procesamiento de declaraciones de variable inicializada (var Id ::= Exp).

  Para esta variable se utilizan como base la generación de código de constantes y de variables ya hechas y se juntaron para crear un tipo de dato nuevo llamado “KnownValueInit” este tipo contiene el tamaño del valor, el nivel, el desplazamiento y el valor inicial de la variable y el tamaño de la dirección, lo que se hace es evaluar la expresión y asignarla al espacio en la pila a la cual corresponde la variable y para las variables cuyo valor se desconoce hasta tiempo de ejecución se utiliza “UnkownValue” de igual manera como se haría una constante.

- Su solución al problema de introducir declaraciones colaterales (par).

  Para par se utiliza la misma solución que con las declaraciones secuenciales.

- Su solución al problema de introducir declaraciones privadas (private).

  Para private se utiliza la misma solución que con las declaraciones secuenciales.

- Su solución al problema de introducir declaraciones de procedimientos o funciones mutuamente recursivos (recursive).

  Para la implementación de recursive se realiza un llamado a el ast ProcoFunc el cual se encarga de realizar el llamado a su contenido permitiendo la recursividad, utilizando las instrucciones de letCommand sin realizar modificaciones.

- Nuevas rutinas de generación o interpretación de código, así como cualquier modificación a las existentes.

- Extensión realizada a los procedimientos o métodos que permiten visualizar la tabla de nombres (aparecen en la pestaña ‘Table Details’ del IDE).

  Se agregó el tipo de dato “KnownValueInit” al tablevisitor para permitir la visualización de variables inicializadas en Table Details dentro del IDE a la hora de de compilar el programa.
- Descripción de la manera en que se resuelve la comprobación de cotas en el acceso a arreglos (en tiempo de ejecución).
- Descripción de cualquier modificación hecha a TAM y a su intérprete.
- Describir cualquier cambio que requirió hacer al analizador sintáctico o al contextual para efectos de generación de código (incluida la representación de árboles sintácticos).
  - Se modifica el analizador contextual para ejecutar correctamente la instrucción de recursive, incluyendo dos variables booleanas las cuales permiten definir cuales visit realizar. Para completar satisfactoriamente estas modificaciones se implementa código proporcionado por los compañeros Marvin Castro Roldán numero de carnet 2017239276 y Ricardo Shum Estrada numero de carnet 2017144193, quienes estuvieron anuentes en compartirlo para así poder implementarlo.

- Describir cualquier modificación hecha al ambiente de programación para hacer más fácil de usar su compilador.

Lista de errores de generación de código o de ejecución detectados

- can't nest routines more than 7 deep
- can't nest routines so deeply
- length of operand can't exceed 255 words
- too many instructions for code segment
- can't access data more than 6 levels out
- can't store values larger than 255 words
- can't load values larger than 255 words

Pruebas positivas

- Debe separar las pruebas para el generador de código de las que validan al intérprete

Objetivo	Diseño	Resultado esperado	Resultado observado
Ver ejecución de if	if 4>3 then puteol() else putint(2) end		0: LOADL 4 1: LOADL 3

			2: CALL gt 3: JUMPIF(0) 6[CB] 4: CALL puteol 5: JUMP 8[CB] 6: LOADL 2 7: CALL putint 8: HALT
Ver ejecución de pass	pass		0: HALT
Ver ejecución de private y par	let private var x ::= 10 in var y ::= x + 11 end; par var p ::= 'a' | var o ::= 'a' end in putint(y); o := 'c' end		0: PUSH 1 1: LOADL 10 2: STORE (1) 0[SB] 3: PUSH 1 4: LOAD (1) 0[SB] 5: LOADL 11 6: CALL add 7: STORE (1) 1[SB] 8: PUSH 1 9: LOADL 97 10: STORE (1) 2[SB] 11: PUSH 1 12: LOADL 97 13: STORE (1) 3[SB] 14: LOAD (1) 1[SB] 15: CALL putint 16: LOADL 99 17: STORE (1) 3[SB] 18: POP (0) 4 19: HALT
Ver ejecución de whileDo	let var i ::= 0 in loop while i < 3 do putint(i); i := i + 1 end end		0: JUMP 6[CB] 1: LOADL 0 2: CALL putint 3: LOADL 0 4: LOADL 1 5: CALL add 6: LOADL 0 7: LOADL 3 8: CALL lt 9: JUMPIF(1) 1[CB] 10: HALT
Ver ejecucion de	let var i:Integer in		0: PUSH 1

untilDo	i := 0;
	loop until i = 3 do
	putint(i);
	put(' ');
	i := i + 1
	end
	end
			1: LOADL 0
			2: STORE (1) 0[SB]
			3: JUMP 12[CB]
			4: LOAD (1) 0[SB]
			5: CALL putint
			6: LOADL 32
			7: CALL put
			8: LOAD (1) 0[SB]
			9: LOADL 1
			10: CALL add
			11: STORE (1) 0[SB]
			12: LOAD (1) 0[SB]
			13: LOADL 3
			14: LOADL 1
			15: CALL eq
			16: JUMPIF(0) 4[CB]
			17: POP (0) 1
			18: HALT

Ver ejecución de DoWhile	let var i:Integer in
	i := 0;
	loop do
	putint(i);
	put(' ');
	i := i + 1
	while i < 3
	end
	end
			0: PUSH 1
			1: LOADL 0
			2: STORE (1) 0[SB]
			3: LOAD (1) 0[SB]
			4: CALL putint
			5: LOADL 32
			6: CALL put
			7: LOAD (1) 0[SB]
			8: LOADL 1
			9: CALL add
			10: STORE (1) 0[SB]
			11: LOAD (1) 0[SB]
			12: LOADL 3
			13: CALL lt
			14: JUMPIF(1) 3[CB]
			15: POP (0) 1
			16: HALT

Ver ejecución de DoUntil	let var i:Integer in
	i := 0;
	loop do
	putint(i);
	put(' ');
	i := i + 1
	until i > 3
	end
	end
			0: PUSH 1
			1: LOADL 0
			2: STORE (1) 0[SB]
			3: LOAD (1) 0[SB]
			4: CALL putint
			5: LOADL 32
			6: CALL put
			7: LOAD (1) 0[SB]
			8: LOADL 1
			9: CALL add
			10: STORE (1) 0[SB]
			11: LOAD (1) 0[SB]

			12: LOADL 3 13: CALL gt 14: JUMPIF(0) 3[CB] 15: POP (0) 1 16: HALT

Ver ejecución de forDo	let var a:Integer in
	loop for i from 0 to 3 do
	a := i +1;
	putint(a);
	put(' ')
	end
	end
		1 2 3 4	0: PUSH 1
			1: LOADL 3
			2: LOADL 0
			3: JUMP 14[CB]
			4: LOAD (1) -1[ST]
			5: LOADL 1
			6: CALL add
			7: STORE (1) 0[SB]
			8: LOAD (1) 0[SB]
			9: CALL putint
			10: LOADL 32
			11: CALL put
			12: POP (1) 0
			13: CALL succ
			14: LOAD (2) -2[ST]
			15: CALL ge
			16: JUMPIF(1) 4[CB]
			17: POP (2) 0
			18: POP (0) 1
			19: HALT

Ver ejecución de forDoUntil	let var a:Integer in
	a := 0;
	loop for i from 0 to 3 until a = 2 do
	a := i + 1;
	putint(a);
	put(' ')
	end
	end
		1 2	0: PUSH 1
			1: LOADL 0
			2: STORE (1) 0[SB]
			3: LOADL 3
			4: LOADL 0
			5: JUMP 16[CB]
			6: LOAD (1) -1[ST]
			7: LOADL 1
			8: CALL add
			9: STORE (1) 0[SB]
			10: LOAD (1) 0[SB]
			11: CALL putint
			12: LOADL 32
			13: CALL put
			14: POP (1) 0
			15: CALL succ
			16: LOAD (2) -2[ST]
			17: CALL ge
			18: LOAD (1) 0[SB]
			19: LOADL 2
			20: LOADL 1
			21: CALL eq

			22: CALL not 23: CALL and 24: JUMPIF(1) 6[CB] 25: POP (2) 0 26: POP (0) 1 27: HALT

Ver ejecución de forDoWhile	let var a:Integer in
	a := 0;
	loop for i from 0 to 3
	while a < 3 do
	a := i + 1;
	putint(a);
	put(' ')
	end
	end
		1 2 3	0: PUSH 1
			1: LOADL 0
			2: STORE (1) 0[SB]
			3: LOADL 3
			4: LOADL 0
			5: JUMP 16[CB]
			6: LOAD (1) -1[ST]
			7: LOADL 1
			8: CALL add
			9: STORE (1) 0[SB]
			10: LOAD (1) 0[SB]
			11: CALL putint
			12: LOADL 32
			13: CALL put
			14: POP (1) 0
			15: CALL succ
			16: LOAD (2) -2[ST]
			17: CALL ge
			18: LOAD (1) 0[SB]
			19: LOADL 3
			20: CALL lt
			21: CALL and
			22: JUMPIF(1) 6[CB]
			23: POP (2) 0
			24: POP (0) 1
			25: HALT

Ver ejecución de assign var	let var n:Integer in
	n := 0;
	loop while n < 3 do
	putint(n);
	put(' ');
	n := n + 1
	end
	end
			0: PUSH 1
			1: LOADL 0
			2: STORE (1) 0[SB]
			3: JUMP 12[CB]
			4: LOAD (1) 0[SB]
			5: CALL putint
			6: LOADL 32
			7: CALL put
			8: LOAD (1) 0[SB]
			9: LOADL 1
			10: CALL add
			11: STORE (1) 0[SB]
			12: LOAD (1) 0[SB]
			13: LOADL 3
			14: CALL lt
			15: JUMPIF(1) 4[CB]
			16: POP (0) 1

			17: HALT

Ver ejecución de ChooseCommand	let
	var a:Integer;
	var b:Integer
	in
	a := 45;
	b := 55;
	choose a from
	when 1
	then
	choose b
	from
	
	when 1
	
	then
	
	putint(33)
	
	when 2
	
	then
	
	putint(34)
	
	else
	
	putint(35)
	
	end
	when 2..3|5
	then
	putint(2)
	else
	putint(3)
	end
	end
		3	0: PUSH 1
			1: PUSH 1
			2: LOADL 45
			3: STORE (1) 0[SB]
			4: LOADL 55
			5: STORE (1) 1[SB]
			6: JUMP 10[CB]
			7: LOADL 2
			8: CALL putint
			9: JUMP 48[CB]
			10: LOAD (1) 0[SB]
			11: LOADL 5
			12: CALL eq
			13: JUMPIF(1) 7[CB]
			14: LOAD (1) 0[SB]
			15: LOADL 2
			16: CALL ge
			17: JUMPIF(0) 42[CB]
			18: LOAD (1) 0[SB]
			19: LOADL 3
			20: CALL le
			21: JUMPIF(1) 7[CB]
			22: JUMP 42[CB]
			23: JUMP 27[CB]
			24: LOADL 34
			25: CALL putint
			26: JUMP 41[CB]
			27: LOAD (1) 1[SB]
			28: LOADL 2
			29: CALL eq
			30: JUMPIF(1) 24[CB]
			31: JUMP 35[CB]
			32: LOADL 33
			33: CALL putint
			34: JUMP 41[CB]
			35: LOAD (1) 1[SB]
			36: LOADL 1
			37: CALL eq
			38: JUMPIF(1) 32[CB]
			39: LOADL 35
			40: CALL putint
			41: JUMP 48[CB]
			42: LOAD (1) 0[SB]
			43: LOADL 1
			44: CALL eq
			45: JUMPIF(1) 23[CB]
			46: LOADL 3

			47: CALL putint 48: POP (0) 2 49: HALT
Ver ejecución de Recursive	let recursive proc x () ~ y() end | proc y () ~ x() end end in pass end		0: JUMP 2[CB] 1: RETURN(0) 0 2: JUMP 5[CB] 3: CALL (SB) 1[CB] 4: RETURN(0) 0 5: HALT

Pruebas Negativas

- Debe separar las pruebas para el generador de código de las que validan al intérprete

Objetivo	Diseño	Resultado esperado	Resultado observado
Ciclo infinito whileDo	let var i ::= 0 in loop while true do putint(i); i := i + 1 end end	0123456789...	0123456789…

Ciclo infinito untilDo	let var i:Integer in
	i := 0;
	loop until false do
	putint(i);
	put(' ');
	i := i + 1
	end
	end
		0 1 2 3 4 5 6 7 8 9 ...	0 1 2 3 4 5 6 7 8 9 ...

Ciclo infinito doUntil	let var i:Integer in
	i := 0;
	loop do
	putint(i);
	put(' ');
	i := i + 1
	until false
	end
	end
		0 1 2 3 4 5 6 7 8 9 ...	0 1 2 3 4 5 6 7 8 9 ...

Ciclo infinito doWhile	let var i:Integer in
	i := 0;
	loop do
	putint(i);
	put(' ');
	i := i + 1
	while true
	end
	end
		0 1 2 3 4 5 6 7 8 9 ...	0 1 2 3 4 5 6 7 8 9 ...

Ciclo forDo	let var a:Integer in
	loop for i from 0 to 1000000000000000000000000000000000000000 do
	a := i +1;
	putint(a);
	put(' ')
	end
	end
		No compile porque el número es muy grande	No compiló porque el número es muy grande

Ciclo forWhileDo	let var a:Integer in
	a := 0;
	loop for i from 0 to 10000000000000 while true do
	a := i + 1;
	putint(a);
	put(' ')
	end
	end
		No compile porque el número es muy grande	No compiló porque el número es muy grande

Ciclo forUntilDo	let var a:Integer in
	a := 0;
	loop for i from 0 to 1000000000000000 until false do
	a := i + 1;
	putint(a);
	put(' ')
	end
	end
		No compile porque el número es muy grande	No compiló porque el número es muy grande

Conclusión

Discusión y análisis de los resultados obtenidos

De manera general se concluye satisfactorias las indicación propuestas por el profesor, permitiéndonos como estudiantes conocer y entender de forma más directa el funcionamiento de un intérprete, al realizar este proyecto el cual se inició con la entrega un mejorando el analizador sintáctico, la entrega dos manejando el analizador contextual y finalmente esta entrega generador de código se nos permite por medio de la experimentación y práctico obtener conocimientos realizar sobre el funcionamiento total de un compilador. Los retos para entender el código ya implementó por terceros fueron grandes en todo momento, pero permitieron adquirir conocimientos, así como una programación más estándar y modulada.

Conclusiones a partir de esto

- Se concluye la implementación de los ciclos while do, until do, do while y do until
- Se concluye la implementación de las instrucciones para los ciclos loop for (do, while y until)
- Se finaliza satisfactoriamente la implementación de choose
- Se implementa correctamente la instrucción de private y par, así como la asignación var.
- Se corrige la instrucción recursive para el analizador contextual.
- Se realizan los casos de pruebas positivos y negativos

Anexos

Tareas realizadas por cada miembro del grupo de trabajo

Tarea realizada	Miembro del grupo
Se implementa las instrucciones de while do, until do, do while, do until	Eduardo
Se implementa las instrucciones de for Control	Eduardo
Se implementa las instrucciones de forDo	Eduardo

Se implementa las instrucciones de private	Andrey
Se implementa las instrucciones de par	Andrey
Se implementa las instrucciones de Choose	Jose Daniel
Se implementa las instrucciones de Var	Andrey
Se implementa las instrucciones de recursive	Kristal
Se implementa las instrucciones de array	
Se implementa las instrucciones de loopForWhile y loopForUntil	Eduardo
Se realizan las pruebas de la instrucción if	Kristal
Se realizan las pruebas de la instrucción pass	Kristal
Se realizan las pruebas de la instrucción while do, until do, do while, do until	Eduardo
Se realizan las pruebas de la instrucción loopForDo	Eduardo
Se realizan las pruebas de la instrucción private	Andrey
Se realizan las pruebas de la instrucción par	Andrey
Se realizan las pruebas de la instrucción choose	José Daniel
Se realizan las pruebas de la instrucción var	Andrey
Se realizan las pruebas de la instrucción recursive	Kristal
Se realizan las pruebas de la instrucción array	
Se realizan las pruebas de la instrucción loopForWhile	Eduardo
Se realizan las pruebas de la instrucción loopForUntil	Eduardo
Documentación	Todos
Corrección en el recursive en el analizador	Kristal

contextual	

Cómo debe compilarse el programa

Abrir netbeans en un sistema operativo windows 7 o superior y seleccionar la opción de abrir proyecto, busca el directorio donde tiene guardado el proyecto y lo abre. Luego selecciona el botón de clean and build para compilar el proyecto.

Cómo debe ejecutarse el programa

Luego de compilado el proyecto se deberá ir a la carpeta dist se habrá creado un archivo llamado IDE-Triangle.jar, se da doble click en el archivo para ejecutar el programa.

Links:

- Enunciado: https://drive.google.com/drive/folders/1wpI4BXHlBPOkDggQQVIAWySLlK62nii4

Referencias

Watt D.A & Brown D.F, 2000, "Programming Language Processing in Java, Compilers and Interpreters"

- Una carpeta que contenga:
  - Carpetas donde se encuentre el texto fuente de sus programas. El texto fuente debe indicar con claridad los puntos en los cuales se han hecho modificaciones.
  - Carpetas donde se encuentre el código objeto del compilador+intérprete, en formato directamente ejecutable desde el sistema operativo Windows22 . Todo el contenido del archivo comprimido debe venir libre de infecciones. Debe incluir el IDE enlazado a su compilador+intérprete, de manera que desde él se pueda ejecutar sus procesadores de y de TAM.
- Debe reunir su trabajo en un archivo comprimido en formato .zip. Esto debe incluir:
  - Documentación indicada arriba, con una portada donde aparezcan los nombres y números de carnet de los miembros del grupo. Los documentos descriptivos deben estar en formato .pdf.
  - Código fuente, organizado en carpetas.
  - Código objeto. Recuerde que el código objeto de su compilador (programa principal) debe estar en un formato directamente ejecutable en Windows.
  - Programas (.tri) de prueba que han preparado.