Surubi
Compilador para Tiger

Carlos David Muñiz Chall
Jesús Manuel Garnica Bonome

14 de abril 2017

1. Descripción

Surubi es un compilador para el lenguaje de programación **Tiger** definido por *Stephen A. Edwards*[2] con la modificaciones requeridas para el año 2017[1]. Este compilador está escrito en el lenguaje de programación *C#*, utilizando un reconocedor sintáctico generado por la herramienta *Antlr 3.4*. A partir del fichero de entrada, el cual es el único argumento requerido, se genera un fichero que contiene código ensamblador listo para ser procesado por el ensamlador de *NASM*.

2. Arquitectura

Durante la compilación interactuan tres componentes principales:

- Reconocedor sintáctico
  Este es el encargado de crear un Ábol de Sintaxis Abstracta(AST) a partir del texto de entrada, la implementación por defecto fue hecha utilizando *Antlr 3.4*
- Comprobador semántico
  Dedicado al chequeo statico de tipos y a la existencia en cada ámbito de los miembros utilizados, la implementación por defecto se llama *DefaultSemanticChecker*.
- Emisor de código
  Provee una interfaz de instrucciones que permiten las acciones usuales en un lenguage de programación, la implementación por defecto *NasmEmitter* produce código *NASM*

La otra parte central en la arquitectura del compilador son los nodos del AST cada uno cuenta con el siguiente conjunto de propiedades y métodos:

- line
  Linea donde fue reconocido el principio de la estructura que representa el nodo.
- column
  Posición en que comienza la estructura en la linea.
- Lex
  Para nodos atómicos, como los literales, es todo el texto asociado con la estructura que representan, en otro caso por lo general no se utiliza.
- CheckSemantics(Comprobador Semantico, Reporte de errores, Tipo esperado)
  Comprueba la correctitud semantica del nodo presente, el valor de retorno determina si el nodo actual pudo decidir el tipo de la etructura que represeta.

- GenerateCode(Emisor de Código, Reporte de errores)
  Utilizando las funciones provistas por el emisor genera código encargado de cumplir con la semantica del nodo.

Esta definición básica es extendida por dos definiciones más especializadas, expresiones y declaraciones. Expresiones, estas representan nodos que después de su ejecución devuelven un valor:

- Return
  Representa el tipo de retorno del valor, este es necesario para el chequeo estatico de tipos.
- ReturnValue
  Representa la información sobre donde va a quedar el valor de retorno durante la ejecución.
- CanBreak
  Valor que indica si el nodo actual puede causar la salida temprana de un ciclo

Las declaraciones representan nodos que sólo añaden miembros al ámbito actual, para manejar las declaraciones cíclicas la comprobación semántica se extendió a dos faces, la fase adicional *BindName* se encarga de añadir la información del miembro que representa, pudiendo estar incompleta, mientras que *CheckSemantics* comprueba la coherencia de esta definición y la completa en caso de ser necesario.

2.1 Reconocedor Sintáctico

Al ser implementado con *Antlr 3.4* este utiliza una gramática *LL(*)*, por lo que las siguientes sintáxis presentan conflictos, donde la alternativa correcta no puede ser decidida por un autómata finito.

lvalue:

id

lvalue . id

lvalue [expr]

lvalue := expr

id (expr-list)

id field-list

id [expr] of expr

Aunque la herramienta utilizada provee facilidades como los predicados sintáticos para resolver este tipo de comflictos, la solución dada solo utiliza factorización izquierda.

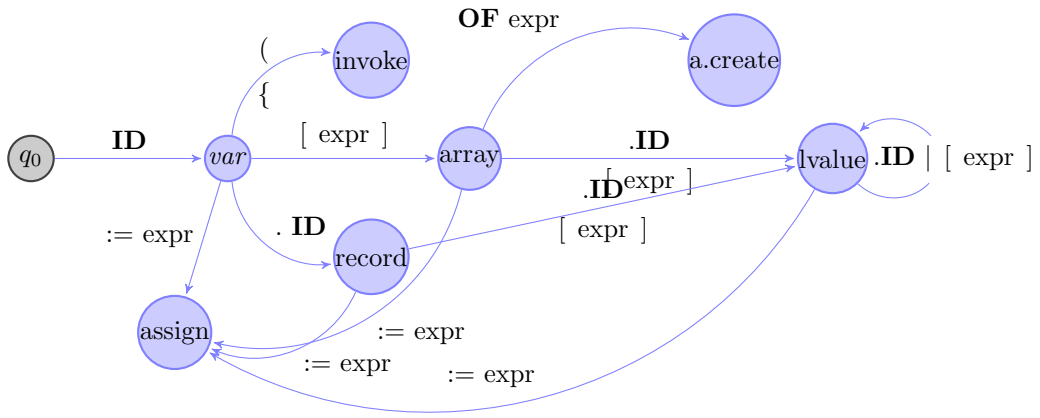

Figura 1: Automata para el reconocimiento de las producciones que comienzan por **ID**

Producciones de la gramatica para lograr el automata anterior.

- lvalue-head:
  ID (invoke | [exp] array | . **ID** record | assign)?
- invoke:
  (argument-list? **)** | { field-list? }
- array:
  OF expr | lvalue
- record:
  lvalue
- lvalue:
  (**.ID** | [exp])* assign?
- assign:
  := expr

El resto de los conflictos fueron operadores binarios del tipo *expr* **op** *expr*, para los cuales se aplicó factorización izquierda reflejando la precedencia de forma explícita.

3. Comprobación Semántica

Durante la comprobación semantica cada nodo chequea a sus descendientes y luego comprueba que el tipo de retorno sea el adecuado. Para manejar el chequeo de los nodos que no devuelven ningún valor se introdujo el tipo *void* y para la constante *nil* el tipo *null*, ninguno de los dos son explícitamente accesibles al código que se está compilando.

Durante este proceso se trabaja con objetos que describen los miembros que serán generados en la siguiente fase.

Miembro

- Name
- BCMMember
  Esta propiedad guarda el menegador generado por la declaración de este miembro, el cual es usado para referirse a el durante la generación de código.
- BCMBackup
  Indica si este miembro requiere de un *BCMMember*, lo cual es el caso de todos los miembros excepto el tipo *void* y el tipo *null*.

Campo, es la abstración que representa una zona de memoria, vaiables y constantes, extiende miembro.

- Type
  Información sobre el tipo del campo.
- Cosnt
  Indica si este miembro es de solo lectura.

Funciones, extiende miembro.

- Parameters
  Lista con el tipo y nombre de los parámetros que recive la función, el orden de la lista coincide con el orden de la declaración y de las invocaciones.
- Return
  Tipo del valor de retorno, si es un procedimiento el tipo es *void*.

Tipos, extiende miembro.

- Members
  Lista con el tipo y nombre de los campos del registro que representa el tipo.
- ArrayOf
  Tipo de los elementos del array que representa el tipo, etsta propiedad y *Members* no deben ser difrenetes de *null* las dos en el mismo tipo.

- Complete
  Indica si la descripción del tipo ha terminado, solo puede ser falsa en fases intermedias de la comprobacion de declaraciones.
- TypeId
  Identificador único para cada tipo, máximo 2^{128} tipos.

3.1 Expresiones

Al terminar la comprobación semántica de un nodo expresión se deben cumplir las siguientes condiciones:

- Si el valor de retorno fue verdadero entonces las propiedades *Return* y *ReturnValue* deben estar correctmente asignadas, *Return* nunca debe ser nula.
- Si y solo si *Return* es el tipo *void* entonces *ReturnValue* es nula.
- Si *CanBreak* es verdadera *Return* es el tipo *void*.
- El valor de retorno es falso si y solo si no se puede determinar el valor de *Return*.

Como consecuencia de esto existen nodos que nunca devuelven falso en su chequeo semántico, como lo son los operadores de enteros los que siempre devuelven *int*, por esta razón el chequeo semántico se considera fallido si se agregan errores al reporte.

3.2 Declaraciones

El chequeo semántico de las declaraciones se dividió en dos fases para manejar las secuencias de declaraciones del mismo tipo. La fase de *BindName* es la primera que se ejecuta excepto en el caso de la declaración de los parámetros de una función. Como las dos fases tienen significados ligeramente diferentes según el tipo de declaración serán expuestas por separado.

En las declaraciones de variables *BindName* no tiene efecto alguno, mientras que *CheckSemantics* hace las comprobaciones y añade el descriptor al ámbito actual. Para los parámetros *CheckSemantics* hace la comprobación estática de tipos y *BindName* añade el descriptor al ámbito actual.

Las funciones se comportan igual que los parámetros pero en orden inverso, para lograr el efecto de las secuencias de declaraciones.

Los tipos usan *BindName* para añadir el descriptor al ámbito actual, solo que si los tipos de los que depende esta declaración no existen el descriptor no es añadido, excepto para los registros los cuales lo añaden marcándolo como incompleto. *CheckSemantics* completa los descriptores y hace los chequeos de tipo, para lo que necesita ser ejecutado después que el chequeo semántico de las declaraciones de la que depende. El orde de ejecución de *CheckSemantics* y de la generación de código en las declaraciones se determina mediante un orden topológico de sus dependencias, comenzando por las que tienen 1 dependencia. Las declaraciones sin dependencias son comprobadas al final.

4. Generación de Código

Para la generación de código se diseño una abstracción del comportamiento de un equipo de cómputo, el conjunto de instrucciones que deben ser implementadas se puede resumir en el siguiente:

- Instrucciones aritméticas.
- Añadir Constantes
- Creación de variables.
- Creación de objetos, registros y vectores.
- Creación de tipos.
- Enlaze con objetos, funciones y tipos de una biblioteca estandar.
- Etiquetas y saltos absolutos y condicionales
- Ámbitos locales y de función
- Emisión de errores
- Invocación de funciones

- Valores de retorno de funciones y parámetros

La implementación se hizo generando ensamblador, la biblioteca estandar está mayormente implentada en ensamblador también con algunas funciones en *C*, para esto fue necesario manejar durante toda la generación dos protocolos de invocación:

- cdcall
  Los parámetros se pasan en la pila de derecha a izquierda y el valor de retorno se deja en *EAX*, el que invoca es el que libera la pila después de la invocación. También utilizado para enlazar con las funciones de la estandar de *C*.

- tiger-call
  Es una extensión de *cdcall* para manejar las clausuras de las funciones de tiger. Cada ámbito de tiger tiene como pimer valor en la pila un puntero al ámbito que lo contiene, para poder utilizar los miembros que declaran sus antecesores. Las funciones de tiger tambien cumplen esto, pero el puntero debe ser almacenado junto con la posición de la función después de su declaración para se pasado como parámetro extremo derecho en cada invocación. Dentro de la función se guarda el puntero del ámbito que la invocó, para ser reestablecido al finalizar esta, y el último parámetro pasa a ser el puntero al ámbito contenedor.

La solución a los saltos hacia posiciones desconocidas es un sistema de reservación de etriquetas que no utiliza *backpatch*, cada nodo que le interece permitir saltos a una etiqueta puesta por él, debe hacerla publica para sus descendientes durante la comprobación semántica, en el caso de los ciclos se utiliza *LoopScopeDescriptor* para que cualquier nodo implementando una salida temprana pueda conocer la etiqueta final de ciclo.

Para obtener un ejecutable a partir de la salida es necesario proveer un ensamblador de *NASM* y un enlazador, junto con el proyecto se provee una carpeta con el ensamblador y el enlazador para *Windows*, el comportamiento por defecto puede ser modificado mediante argumentos en el comando de entrada.

Referencias

[1] Colectivo de Compilación. MATCOM. UH. Tiger proyecto de compilación. 2016-2017.

[2] Stephen A. Edwards. Tiger language reference manual.