Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores
Febrero-abril 2022

Etapa II: Análisis Sintáctico con construcción de Árbol Sintáctico Abstracto

En esta entrega ustedes deberán especificar la sintaxis del lenguaje Stókhos por medio de una gramática libre de contexto e implementar un analizador sintáctico (reconocedor) del lenguaje basado en dicha gramática. De hecho, van a especificar la sintaxis con dos gramáticas: una para documentación y otra para la implementación del reconocedor. Dicho reconocedor, además de validar la sintaxis de la entrada, deberá construir el árbol sintáctico abstracto (AST) de cada expresión o instrucción indicada por el usuario. Para validar el reconocimiento correcto del lenguaje Stókhos, la VM permite visualizar el AST como una secuencia de caracteres.

El reconocedor debe aceptar toda instrucción o expresión *sintácticamente* valida de Stókhos, pero no tiene que validar la *semántica*, lo cual será el objetivo de la tercera etapa. Por ejemplo, no es necesario validar que una variable utilizada en una expresión haya sido declarada previamente ni hacer chequeos de consistencia de tipos.

1. Especificaciones para la entrega

1.1 Gramática

Cada equipo debe especificar la sintaxis de Stókhos por medio de dos gramáticas libre de contexto, descritas en un archivo (gramatica.md) en el repositorio del proyecto. La primera gramática debe describir la sintaxis de Stókhos de manera natural e intuitiva, pero rigurosa. Para inspiración, aquí se presenta un posible fragmento:

```
<entrada> -> <instrucción> | <expresión>

<instrucción> -> <definición> | <asignación>

<definición> -> <tipo> <identificador> := <expresión> ;

        …

<expresión>

        -> <número>

                …

        -> <expresión> + <expresión>
```

Los símbolos no terminales (y pseudo-terminales) se especifican con un identificador “descriptivo” encerrado por `< >`; por ejemplo, `<entrada>` se refiere a lo tecleado por el usuario en el REPL – es claramente el símbolo de arranque - y el lado derecho de sus producciones (nótese que son dos producciones, abreviadas en una línea) indican que la `<entrada>` puede ser una `<instrucción>` o una `<expresión>`. Esta es la gramática que ***Uds.****, programadores*, desean ver en un manual de un lenguaje: ¡más “natural e intuitiva” no se puede!

La segunda gramática (transformada) corresponde a la utilizada para la construcción del reconocedor, bien sea que la construcción es realizada por un generador (aconsejable) o "manualmente", siguiendo las reglas de construcción de reconocedores a partir de gramáticas.

La razón por la cual es necesario especificar la gramática de dos maneras es que, dependiendo de la técnica de reconocimiento usada (LL, LR, PEG, …) y los detalles del generador, es posible que algunos equipos tengan que aplicar varias transformaciones a la primera gramática para poder generar el reconocedor. Obviamente, dichas transformaciones deben ser realizadas con cuidado: la segunda gramática debe ser ***equivalente*** a la primera, de otra forma el reconocedor no va a ser consistente con su entendimiento del lenguaje. La intención no es darles más trabajo, sino ayudarles a pensar de manera clara y rigurosa. En particular:

1) Si la herramienta usada por Uds. permite especificar una gramática de expresiones ambigua, con resolución de ambigüedad garantizada por las reglas de precedencia y asociatividad de operadores, se vale usar esa gramática, *anexando la especificación de las reglas de precedencia y asociatividad al documento*. Esto permite especificar la sintaxis de manera natural y precisa.
2) Independientemente de la metodología o herramienta usada para especificar la segunda gramática, se vale abreviar las partes que son iguales, diciendo "ver gramática anterior".

1.2 Reconocedor (Parser)

Como es de esperarse, su reconocedor sintáctico utilizará el analizador lexicográfico construido en la primera etapa. Si su analizador lexicográfico tiene defectos, corríjanlos ya o hablen con el preparador si necesitan ayuda.

En la VM, deben implementar una función llamada ***parse***. La función ***parse*** recibe la secuencia de caracteres correspondiente a la entrada indicada por el usuario y retorna el AST correspondiente. El AST es una estructura interna que depende del lenguaje usado para implementar a Stókhos y de sus preferencias de modelamiento, pero dicha estructura debe incluir toda la información necesaria para validar la semántica e implementarla.

Obviamente ***parse*** (2da entrega) internamente invoca al ***lexer*** (1ra entrega). Nótese que el uso del nombre ***parse*** es *obligatorio* para ayudar a los asistentes de laboratorio a conseguir la función.

Para probar el reconocedor debemos permitir que el REPL llame a ***parse***: otra violación del principio de minimizar la interfaz de la VM, justificada en términos de enseñanza y validación del proyecto. Sin embargo, hay un problema: la función ***parse*** retorna un AST, una estructura intima de la VM, no un string que el REPL puede imprimir directamente. La solución es no llamar a ***parse*** directamente, sino a una función en la VM que llama a ***parse*** y convierte el AST resultante en un string que puede ser consumido por el REPL. Vamos a darle un nombre obligatorio a esa función: ***testParser***. La función ***testParser*** es parte de la interfaz de la VM: primero llama a ***parse***, para obtener el AST, y luego a ***ast2str***, otra función de la VM, para convertir el AST en una secuencia de caracteres. Técnicamente, ***ast2str*** *implementa una traducción*. Para simplificar la traducción, y hacerla amigable, las expresiones deben ser regeneradas con paréntesis redundantes usando notación infija:

"Alpha * (Beta + Gamma)" =*parse*=> [[AST]] =*ast2str* => "(Alpha * (Beta + Gamma))"

"num x := 6 * 7;" =*parse*=> [[AST]] =*ast2str*=> "def(num, x, (6 * 7))"

Como pueden apreciar, la representación de una definición provee los ingredientes esenciales para su futura interpretación. Para finalizar esta etapa, deben agregar el comando especial ***.ast*** al REPL:

.ast <entrada>

lógicamente, ***.ast*** hace que REPL llame a ***testParser*** en vez de ***process***, pasando la entrada como argumento. Como ven, tenemos tres funciones en la interfaz de la VM hasta la fecha: *lexTest*, *testParser*, y *process*.