Universidad Nacional de Río Cuarto
Facultad de Ciencias Exactas Físico-Químicas y Naturales
Departamento de Computación

Taller de Diseño de Software

(Cod. 3306)

Descripción del Lenguaje: `MINI-TDS`

2017

El proyecto de la materia consiste en implementar un compilador para un lenguaje imperativo simple, similar a `C` o `Pascal`, llamado `MINI-TDS`.

Consideraciones del Léxico

El lenguaje `MINI-TDS` es *case-sensitive*. Las **palabras reservadas** del lenguaje únicamente están formadas por minúsculas. Las palabras reservadas y los identificadores son *case-sensitive*. Por ejemplo, **while** es una palabra reservada, pero `WHILE` es un identificador; `cont` and `Cont` son dos nombres diferentes de variables distintas.

Las palabras reservadas son:

class if else then false true void int bool return while

Dos tipos de comentarios son permitidos, aquellos que comienzan con `//` y terminan al final de la línea (únicamente pueden ser de una línea) y aquellos que están delimitados por `/*` y `*/` (pueden tener varias líneas de extensión).

Uno o más espacios pueden aparecer entre los símbolos del lenguaje. Llamamos espacios a los espacios en blanco, tabulaciones, saltos de líneas y/o comentarios.

Las palabras reservadas y los identificadores deben estar separados por un espacio o por un símbolo que no es ni una palabra reservada ni un identificador. Por ejemplo, `whiletrue` es un identificador, no dos palabras reservadas.

Los literales del lenguaje son: números enteros. Los literales enteros son iguales a los utilizados en `C` (por ejemplo 123). Los números enteros son de 32 bit con signo, es decir, los valores están en el rango entre -2147483648 y 2147483647.

Gramática

Notación:

⟨simb⟩	⟨simb⟩ es un no-terminal.
simb	**simb** es un terminal
$\left[x\right]$	cero o una ocurrencia de x, *i.e.,* x es opcional; notar que corchetes entre comillas '[' ']' son terminales.
x^*	cero o más ocurrencias de x.
x^+**,**	una lista de una o más ocurrencias de x's separadas por coma.
$\left\{\ \right\}$	llaves son usadas para agrupar; notar que llaves entre comillas '{' '}' son terminales.
|	separa alternativas.

⟨program⟩ → **class** '{' ⟨var_decl⟩* ⟨method_decl⟩* '}'

⟨var_decl⟩ → ⟨type⟩ $\left\{\text{⟨id⟩}\right\}^+$**,** ;

⟨method_decl⟩ → $\left\{\text{⟨type⟩} \mid \textbf{void}\right\}$ ⟨id⟩ ($\left[\left\{\text{⟨type⟩ ⟨id⟩}\right\}^+\text{,}\right]$) ⟨block⟩

⟨block⟩ → '{' ⟨var_decl⟩* ⟨statement⟩* '}'

⟨type⟩ → **int** | **bool**

⟨statement⟩ → **if** (⟨expr⟩) **then** ⟨block⟩ [**else** ⟨block⟩]
| **while** ⟨expr⟩ ⟨block⟩
| **return** [⟨expr⟩] ;
| ⟨id⟩ = ⟨expr⟩ ;
| ⟨method_call⟩ ;
| ;
| ⟨block⟩

⟨method_call⟩ → ⟨id⟩ ($\left[\text{⟨expr⟩}^+\text{,}\right]$)

⟨expr⟩ → ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩
| - ⟨expr⟩
| ! ⟨expr⟩
| (⟨expr⟩)
| ⟨id⟩
| ⟨method_call⟩
| ⟨literal⟩

⟨bin_op⟩ → ⟨arith_op⟩ | ⟨rel_op⟩ | ⟨cond_op⟩

⟨arith_op⟩ → `+` | `-` | `*` | `/` | `%`

⟨rel_op⟩ → `<` | `>` | `==`

⟨cond_op⟩ → `&&` | `||`

⟨literal⟩ → ⟨int_literal⟩ | ⟨bool_literal⟩

⟨id⟩ → ⟨alpha⟩ ⟨alpha_num⟩*

⟨alpha_num⟩ → ⟨alpha⟩ | ⟨digit⟩ | _

⟨alpha⟩ → `a` | `b` | `...` | `z` | `A` | `B` | `...` | `Z`

⟨digit⟩ → `0` | `1` | `2` | `...` | `9`

⟨int_literal⟩ → ⟨digit⟩ ⟨digit⟩*

⟨bool_literal⟩ → **true** | **false**

Ejemplo de Programa en MINI-TDS

```
class {
  int inc(int x) {
      return x + 1;
  }
  void main() {
      int y;
      y = 4;
      if (y == 1) then {
          return 1;
      } else {
          return inc(y);
      }
  }
}
```

Semántica

Un programa MINI-TDS consiste de una lista de declaraciones de variables globales y funciones. El programa debe contener la declaración de un método llamado **main**. Este método no tiene parametros. La ejecución de un programa MINI-TDS comienza con el método **main**.

Tipos

Los tipos básicos en MINI-TDS son **int** y **bool**.

Reglas de Alcance y Visibilidad de los Identificadores

Las reglas de alcance y visibilidad en MINI-TDS son simples. Primero, todos los identificadores deben ser definidos (textualmente) antes de ser usados. Por ejemplo, una variable debe ser declarada antes de ser usada; una función puede ser invocada únicamente por código ubicado después de su declaración.

En un punto de un programa MINI-TDS existen al menos dos ámbitos (*scopes*) válidos, el global y el local a la función. El scope global esta conformado por los identificadores de las variables y de las funciones declaradas al definir el programa. El scope de la función esta conformado por los parámetros formales y los identificadores de las variables declaradas en el cuerpo de la función. Se pueden definir scope locales adicionales al introducir bloques (⟨block⟩) de código. Los distintos scopes tienen una relación de anidamiento, tal que, el scope global contiene a el scope de las funciones y estos contienen a los scopes de los bloques (los cuales tambien pueden ser declarados de manera anidada). Este anidamiento causa que identificadores definidos en un scope pueda ocultar un identificador con el mismo nombre en scopes superiores. Se debe notar que una variable local puede ocultar tanto un identificador de una variable como de una función.

Los nombres de los identificadores son únicos en cada scope. Es decir, no se puede utilizar el mismo identificador más de una vez en cada ámbito. Por ejemplo, variables y funciones deben tener distinto nombre en el scope global.

Locaciones en Memoria

El lenguaje MINI-TDS tiene una clase de locaciones: variables (locales y globales). Cada locación en memoria tiene un tipo. Por ejemplo, las locaciones de tipo **int** y **bool** contienen valores enteros y lógicos, respectivamente;

Cada locación es inicializada con un valor por defecto cuando es declarada. Los enteros son inicializados con cero y los boleanos con **false**. Se debe notar que esto implica que cada variable local es inicializada cada vez que se entra al bloque en el que se declara.

Asignaciones

Solo se permiten asignaciones a variables de tipos básicos, es decir, variables de tipos **int** y **bool**. La semántica de las asignaciones define la copia del valor. La asignación ⟨location⟩ = ⟨expr⟩ copia el valor resultante de evaluar la ⟨expr⟩ en ⟨location⟩ (copia el valor de la expresión en la variable). Una asignación es válida si ⟨location⟩ y ⟨expr⟩ tienen el mismo tipo. Las asignaciones de incremento y decremento únicamente son permitidas para tipos numéricos.

Se permite asignar valores a los parámetros de un método, pero el efecto de estas asignaciones únicamente es visible en el scope del método. Los parámetros son pasados por valor.

Invocación y Retorno de Métodos

La invocación de métodos involucra: (1) pasar los valores de los parámetros reales del método que invoca al invocado; (2) ejecutar el cuerpo del método invocado; y (3) retornar del método invocado, posiblemente retornando un resultado.

Los argumentos son pasados por valor. El valor de evaluar los parámetros reales es copiado a los parámetros formales, los cuales, son considerados como variables locales del método. Los parámetros son evaluados de izquierda a derecha.

Un método que no tiene declarado un tipo de retorno (un método **void**) únicamente puede ser invocado como una sentencia, es decir, no puede ser usado como una expresión. Estos métodos retornan con una sentencia **return** (sin expresión) o cuando el fin del método es alcanzado.

Un método que retorna un resultado puede ser invocado como parte de una expresión. Estos métodos no pueden alcanzar el fin del método, es decir, únicamente retornan con una sentencia **return** (que debe tener asociado una expresión).

Un método que retorna un resultado también puede ser invocado como una sentencia. En este caso, el resultado es ignorado.

Sentencias de Control

if. La sentencia **if** tiene la semántica estándar. Primero, la ⟨expr⟩ es evaluada. Si el resultado es **true**, la rama del *then* es ejecutada. En otro caso, se ejecuta la rama del **else**, si existe.

while. La sentencia **while** tiene la semántica estándar. Primero, la ⟨expr⟩ es evaluada. Si el resultado es **false**, el cuerpo del ciclo no se ejecuta. En otro caso, el cuerpo del ciclo es ejecutado. Al terminar de ejecutar el cuerpo del ciclo, la sentencia **while** es ejecutada nuevamente.

Expresiones. Las expresiones siguen las reglas usuales de evaluación. En ausencia de otras restricciones, los operadores con la misma precedecia son evaluados de izquierda a derecha. Los paréntesis pueden ser usados para modificar la precedencia usual.

Una locación (variables) son evaluados al valor que contiene la locación en memoria.

Literales enteros se evalúan a su valor.

Los operadores aritméticos (⟨arith_op⟩ y menos unario) y los operadores relacionales (⟨rel_op⟩) tienen el significado y precedencia usual. `%` computa el resto de una división de números enteros.

Los operadores relacionales son usados para comparar expresiones númericas. El operador de igualdad (`==`, *es igual* son definidos para todos los tipos básicos. Únicamente se pueden comparar expresiones del mismo tipo.

El resultado de un operador relacional o de igualdad tienen tipo **bool**.

Los operadores lógicos `&&` y `||` deben ser evaluados usando evaluación de *corto circuito*. El segundo operador no es evaluado si el primer operador determina el valor de toda la expresión, es decir, si el resultado es **false** para `&&` o **true** para `||`.

Precedencia de operadores, de mayor precedencia a menor precedencia:

Operadores	*Comentarios*
`-`	menos unario
`!`	negación lógica
`* / %`	multiplicación, división, resto
`+ -`	suma, resta
`< >`	relacionales (menor, mayor)
`==`	igual
`&&`	conjunción (and)
`||`	disyunción (or)

Notar que estas reglas de precedencia no esta reflejada en la gramática.

Reglas Semánticas

Estas reglas son restricciones (semánticas) adicionales a las reglas sintácticas expresadas en la gramática. Un programa es válido si esta bien formado gramaticálmente y no viola ninguna de las siguientes reglas. El compilador deberá verificar estas reglas, en caso de detectar que no se cumple alguna, deberá generar un mensaje de error que describa el error detectado. Si el compilador no detecta ninguna violación no debera generar ningún informe.

1. Ningún identificador es declarado dos veces en un mismo bloque.
2. Ningún identificador es usado antes de ser declarado.

3. Todo programa contiene la definición de un método llamado **main**. Este método no tiene parámetros. Notar que la ejecución comienza con el método **main**.
4. El número y tipos de los argumentos en una invocación a un método debe ser iguales al número y tipos declarados en la definición del método (los parámetros formales y los reales deben ser iguales).
5. Si la invocación a un método es usada como una expresión, el método debe retornar un resultado.
6. Una sentencia **return** solo tiene asociada una expresión si el método retorna un valor, si le método no retorna un valor (es un método **void**) entonces la sentencia **return** no puede tener asociada ninguna expresión.
7. La expresión en una sentencia **return** debe ser igual al tipo de retorno declarado para el método.
8. Un ⟨id⟩ usado como una ⟨location⟩ debe estar declarado como un parámetro o como una variable local o global.
9. La ⟨expr⟩ en una sentencia **if** o **while** debe ser **bool**.
10. Los operandos de ⟨arith_op⟩'s y ⟨rel_op⟩'s deben ser de tipo **int**.
11. Los operandos de ⟨eq_op⟩'s (`==`) deben tener el mismo tipo (**int** o **bool**).
12. Los operandos de ⟨cond_op⟩'s y el operando de la negación (**!**) deben ser de tipo **bool**.
13. La ⟨location⟩ y la ⟨expr⟩ en una asignación, ⟨location⟩ `=` ⟨expr⟩, deben tener el mismo tipo.