Universidad Nacional de Rio Cuarto
Facultad de Ciencias Exactas
Departamento de Computacion
Taller de Diseño de Software
Prof. Francisco Bavera, Cecilia Kilmurray

Informe

Baseggio, Axel Gabriel
Noviembre 2022

Lexico:
El lenguaje TDS22 es *case-sensitive*. Las **palabras reservadas** del lenguaje unicamente estan formadas por minusculas. Las palabras reservadas y los identificadores son *case-sensitive*. Por ejemplo, **while** es una palabra reservada, pero **WHILE** es un identificador; cont y Cont son dos nombres diferentes de variables distintas.

Las palabras reservadas son:
extern bool program else then false if int return true void while

Dos tipos de comentarios son permitidos, aquellos que estan delimitados por /* y */.
Uno o mas espacios pueden aparecer entre los simbolos del lenguaje. Llamamos espacios a los espacios en blanco, tabulaciones, saltos de lines y/o comentarios.

Las palabras reservadas y los identificadores deben estar separados por un espacio o por un simbolo 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: numeros enteros. Los literales enteros son iguales a los utilizados en C (por ejemplo 123). Los numeros enteros son de 32 bit con signo, es decir, los valores estan en el rango entre -2147483648 y 2147483647.

Consideraciones del Lexico:
No todas las palabras reservadas estan formadas unicamente por minusculas, por ejemplo la palabra reservada **program** se cambio por **Program**. Y se cambio el nombre a algunas palabras reservadas.

Cambios en las palabras reservadas:

Antes	**Despues**
program	Program
bool	Bool
integer	Int
true	True

false	False

Gramatica:
Gramatica original:
<program> -> **program** ‘{’ <var_decl>* <method_decl>* ‘}’
<var_decl> -> <type> <id> = <expr> ;
<method_decl> -> { <type> | void } <id> ([{<type> <id>}+,]) <block>
-> { <type> | void } <id> ([{<type> <id>}+,]) **extern** ;

<block> -> ‘{‘ <var_decl>* <statement>* ’}’
<type> -> **integer** | **bool**

<statement> -> <id> = <expr> ;
| <method_call> ;
| **if** (<expr>) **then** <block> [**else** <block>]
| **while** <expr> <block>
| **return** [<expr>] ;
| ;
| <block>

<method_call> -> <id> ([<expr>+,])

<expr> -> <id>
| <method_call>
| <literal>
| <expr> <bin_op> <expr>
| - <expr>
| ! <expr>
| (<expr>)

<bin_op> -> <arith_op> | <rel_op> | <cond_op>
<arith_op> -> + | - | * | / | %
<rel_op> -> < | > | ==
<cond_op> -> && | ||

<literal> -> <integer_literal> | <bool_literal>
<id> -> <alpha> <alpha_num>*
<alpha> -> a | b | … | z | A | B | … | Z
<alpha_num> -> <alpha> | <digit> | _
<digit> -> 0 | 1 | 2 | … | 9
<integer_literal> -> <digit> <digit>*
<bool_literal> -> **true** | **false**

Gramatica nueva:
<program> -> **program** ‘{’ <declarations> <method_decls> ‘}’

<declarations> -> λ

| <declarations> <declaration>
<declaration> -> <type> <id> = <expr> ;

<method_decls> -> λ
| <method_decl> <method_decls>
<method_decl> -> <method> <block>
| <method> **extern** ;

<method> -> **void** <id> ‘(’ parameters ‘)’
| <type> <id> ‘(’ parameters ’)’

<block> -> ‘{’ <declarations> <statements> ‘}’

<id> -> <alpha> <alpha_num>*

<alpha> -> a | b | … | z | A | B | … | Z
<alpha_num> -> <alpha> | <digit> | _
<digit> -> 0 | 1 | 2 | … | 9

<type> -> **Int** | **Bool**

<parameters> -> λ
| <one_or_more_parameters>

<one_or_more_parameters> -> <parameter>
| <parameter> ‘,’ <one_or_more_parameters>

<parameter> -> <type> <id>

<statements> -> λ
| <statements> <statement>

<statement> -> <id> ‘=’ <expr>
| <expr> ‘;’
| **if** '(' <expr> ')' **then** <block>
| **if** '(' <expr> ')' **then** <block> **else** <block>
| **while** <expr> <block>
| **return** ‘;’
| **return** <expr> ‘;’
| ‘;’
| <block>
<expr> -> <id>
| <method_call>
| <value>
| <expr> ‘+’ <expr>
| <expr> ‘-’ <expr>
| <expr> ‘*’ <expr>
| <expr> ‘/’ <expr>

| <expr> ‘%’ <expr>
| <expr> ‘<’ <expr>
| <expr> ‘>’ <expr>
| <expr> ‘==’ <expr>
| <expr> ‘||’ <expr>
| <expr> ‘&&’ <expr>
| ‘-’ <expr>
| ‘!’ <expr>
| ‘(’ <expr> ‘)’

<value> -> <int_literal> | <bool_literal>
<int_literal> -> <digit> <digit>*
<bool_literal> -> **True** | **False**
<method_call> -> <id> ‘(’ <exprs> ‘)’
<exprs> -> λ
| <one_or_more_exprs>

<one_or_more_exprs> -> <expr>
| <expr> ‘,’ <one_or_more_exprs>

Semantica
Un program TDS22 consiste de una lista de declaraciones de variables globales y funciones. El programa debe contener la declaracion de un metodo llamado **main**. Este metodo no tiene parametros. La ejecucion de un programa TDS22 comienza con el metodo **main**.

Se permite a nivel sintactico que el metodo **main** tenga parametros pero se desechan.

Ejemplo de un programa sin metodo **main**:

```
Program {

}
```

Se informara el error como:
main function is not defined

Reglas de Alcance y Visibilidad de los Identificadores
Todos los identificadores deben ser definidos antes de ser usados.

Ejemplo:

```
Program {
	Int main() {
		x = x + 1;
		return 0;
	}
}
```

Se informara el error como:
Undeclared identifier: x

Una funcion puede ser invocada por codigo ubicado despues de su declaracion.

Ejemplo;

```
Program {
	Int f() {
		return 1 + g(0);
	}

	Int g(Int x) {
		return x;
	}

	Int main() {
		return 0;
	}
}
```

Se informara el error como:
Undeclared identifier: g

En un punto de un programa TDS22 existen al menos dos ambitos (scopes) validos, el global y el local a la funcion. El scope global esta conformado por los identificadores de las variables y de las funciones declaradas al definir el programa. El scope de la funcion esta conformado por los parametros formales y los identificadores de las variables declaradas en el cuerpo de la funcion.

Ejemplo:

```
Program {
	Int x = 0;
	void print(Int x) extern;
	Int main() {
		Int x = 1;
		print(x);
		return 0;
	}
}
```

Output: 1

Se puede definir scope locales adicionales al introducir bloques de codigo. Los distintos scopes tienen relacion de anidamiento, tal que, el scope global contiene al scope de las funciones y estos contienen a los scopes de los bloques. Este anidamiento cause que

identificadores definidos en un scope puedan ocultar un identificador con el mismo nombre en scopes superiores.

Ejemplo:

```
Program {
	Int x = 0;

	void print(Int x) extern;

	void main() {
		{
			Int x = 1;
			{
				Int x = 2;
				print(x);
			}
			print(x);
		}
		print(x);
	}
}
```

Output:
2
1
0

Tambien en un bloque de una funcion se puede ocultar un parametro, ejemplo:

```
Program {
	void print(Int x) extern;
	void f(Int x) {
		Int x = 0;
		print(x);
	}
	void main() {
		f(1);
	}
}
```

Output:
0

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

```
Program {
	Int x = 0;

	Int x(Int y, Int z) {
		return y + z;
	}

	void main() {

	}
}
```

Se informara el error como:
Redeclared identifier: x

Se permite lo siguiente:

```
Program {
	Int x(Int x) {
		return x;
	}

	void main() {

	}
}
```

Asignaciones
Solo se permiten asignaciones a variables de tipos basicos, es decir, variables de tipos **Int** y **Bool**. La semantica de las asignaciones define la copia del valor. La asignacion <location> = <expr> copia el valor resultante de evaluar la <expr> en <location>. Una asignacion es valida si <location> y <expr> tienen el mismo tipo.

Ejemplo:

```
Program {
	void main() {
		Int x = True;
	}
}
```

Se informara el error como:
x is of type Int but the expression is of type Bool

Se permite asignar valores a los parametros de un metodo pero el efecto de estas asignaciones unicamente es visible en el scope del metodo. Los parametros son pasados por valor.

Ejemplo:

```
Program {
	void f(Int x) {
		x = 1;
	}

	void print(Int x) extern;

	void main() {
		Int x = 0;
		f(x);
		print(x);
	}
}
```

Output:
0

Invocacion y Retorno de Metodos
Un metodo que no tiene declarado un tipo de retorno (un metodo **void**) unicamente puede ser invocado como una sentencia, es decir, no puede ser usado como una expresion. Estos metodos retornan una sentencia **return** (sin expresion) o cuando el fin del metodo es alcanzado.

Ejemplo:
```
Program {
	void f(Int x) {

	}

	void main() {
		Int x = 1 + f(1);
	}
}
```

Se informara el error como:
+ arguments are of type Int x Int but Int x void were found

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

Ejemplo:
```
Program {
	Int fib(Int n) {
		if (n == 0) then {
			return 0;
		}
```

```
        if (n == 1) then {
            return 1;
        }

        return fib(n - 1) + fib(n - 2);
    }

    void print(Int x) extern;

    void main() {
        print(fib(10));
    }
}
```

Output: 55

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

Sentencias de Control
if. La sentencia **if** tiene la semantica estandar. 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 semantica estandar. 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 el cuerpo del ciclo, la sentencia **while** es ejecutada nuevamente.

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

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

Literales enteros se evaluan a su valor.

Los operadores aritmeticos y los operadores relacionales tiene el significado y precedencia usual.

Los operadores relaciones son usados para comparar expresiones numericas. El operador de igualdad es definido para todos los tipos basicos. Unicamente se pueden comparar expresiones del mismo tipo.

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

Los operadores logicos && y || deben ser evaluados usando evaluacion de corto circuito.

Llamadas a librerias. El lenguaje TDS22 incluye un mecanismo similar al provisto por el lenguaje C para invocar metodos de librerias externas. Las funciones externas se definen utilizando la palabra reservada **extern**.

Consideraciones extra:

- No se puede asignar un valor a una funcion.

```
Program {
        Int f(Int x) {
                return x;
        }
        Int main() {
                f = 0;
                return 0;
        }
}
```

  Se informara el error como:
  ERROR: f is a function.

- No se puede retornar un identificador correspondiente a una funcion de la siguiente forma:

```
Program {
        Int zero() {
                return 0;
        }

        Int f() {
                return zero;
        }

        void main() {
                f();
        }
}
```

  Se informara el error como:
  ERROR: zero is a function.

- Se permite a nivel sintactico tener mas de 6 parametros pero el comportamiento del programa es indefinido si se utiliza algun parametro desde el septimo en adelante.
- Unicamente se puede inicializar una variable global con expresiones formadas por constantes.

```
Program {
        Int f  = 0;
        Int x = f + 1;

        void main() {

        }
}
```

Se informara el error como:
error: initializer element is not constant

- No se puede utilizar una variable como si fuese una funcion.

```
Program {
        Int x = 0;
        void main() {
                x();
        }
}
```

Se informara el error como:
x is not a function

Con respecto al chequeo de parametros en la llamada a una funcion se chequea que:

- El numero de parametros formales y reales coincidan.
- El tipo de parametros formales y reales coincidan.

Ejemplo:

```
Program {
        Int f(Int x, Int y) {
                return x + y - y;
        }

        void main() {
                f(0);
        }
}
```

Se informara el error como:
too few params to function f

```
Program {
        Int f(Int x, Int y) {
                return x + y - y;
        }

        void main() {
                f(0, 1, 2);
        }
}
```

Se informara el error como:
too many params to function f

```
Program {
        Int f(Int x, Int y) {
                return x + y - y;
        }
```

```
        void main() {
                f(True, 1);
        }
}
```

Se informara el error como:
function f expected Int but argument is of type Bool

```
Program {
        Int f(Int x, Int y) {
                return x + y - y;
        }

        void main() {
                f(0, False);
        }
}
```

Se informara el error como:
function f expected Int but argument is of type Bool

Con respecto al chequeo del return de una funcion se chequea:

- Si la funcion tiene al menos un return que el tipo de todos los return que tenga sea el mismo que el tipo de la funcion.
- Una funcion void no tiene return, excepto si es un return con una expresion vacia.

Ejemplo:

```
Program {
        void main() {
                return 0;
        }
}
```

Se informara el error como:
void was expected in return statement in main function

Nota: un return ; tiene como tipo void.

Generacion de codigo intermedio:

Optimizaciones:
Las optimizaciones que se implementaron fueron:

- Propagacion de constantes:
  Ejemplo:

  ```
  Program {
          Int main() {
  ```

```
                    Int x = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10;
                    return 0;
                }
        }
```

Codigo Assembler sin propagacion de constantes:

```
    .text
    .globl main
main:
    enter $(32), $0
    movq $1, %r10
    imulq $2, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $3, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $4, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $5, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $6, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $7, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $8, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $9, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $10, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    movq %r10, -8(%rbp)
    movq $0, %rax
    leave
    ret
    leave
    ret
```

Codigo Assembler con propagacion de constantes:

```
    .text
    .globl main
```

```
main:
    enter $(16), $0
    movq $3628800, %r10
    movq %r10, -8(%rbp)
    movq $0, %rax
    leave
    ret
    leave
    ret
```

- Reutilizacion de offsets:
  Ejemplo:

  Codigo Assembler sin reutilizacion:

```
        .text
    .globl main
main:
    enter $(96), $0
    movq $1, %r10
    imulq $2, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $3, %r10
    movq %r10, -24(%rbp)
    movq -24(%rbp), %r10
    imulq $4, %r10
    movq %r10, -32(%rbp)
    movq -32(%rbp), %r10
    imulq $5, %r10
    movq %r10, -40(%rbp)
    movq -40(%rbp), %r10
    imulq $6, %r10
    movq %r10, -48(%rbp)
    movq -48(%rbp), %r10
    imulq $7, %r10
    movq %r10, -56(%rbp)
    movq -56(%rbp), %r10
    imulq $8, %r10
    movq %r10, -64(%rbp)
    movq -64(%rbp), %r10
    imulq $9, %r10
    movq %r10, -72(%rbp)
    movq -72(%rbp), %r10
    imulq $10, %r10
    movq %r10, -80(%rbp)
    movq -80(%rbp), %r10
    movq %r10, -8(%rbp)
    movq $0, %rax
```

```
    leave
    ret
    leave
    ret
```

Codigo Assembler con reutilizacion:

```
        .text
    .globl main
main:
    enter $(32), $0
    movq $1, %r10
    imulq $2, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $3, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $4, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $5, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $6, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $7, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $8, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $9, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    imulq $10, %r10
    movq %r10, -16(%rbp)
    movq -16(%rbp), %r10
    movq %r10, -8(%rbp)
    movq $0, %rax
    leave
    ret
    leave
    ret
```