Compiladores II

Definición del Lenguaje `MiniLPP`

Iván de Jesús Deras Tábora

Universidad Tecnlógica Centroamericana (UNITEC)

07/Mayo/2019

1 Introducción

El proyecto de la clase consite en implementar un compilador para un lenguaje llamado MiniLPP. El cual es un lenguaje fuertemente tipificado, con soporte para tipos de datos enteros, booleanos, caracter y arreglos unidimensionales. MiniLPP es un subconjunto del lenguaje LPP. Debemos tener en cuenta que MiniLPP no es una copia exacta del lenguaje LPP. La funcionalidad de MiniLPP ha sido reducida considerablemente comparado con un lenguaje "completo". Esto se hizo para hacer el proyecto de la clase realizable en un trimestre (10 semanas). A pesar de esto MiniLPP será capaz de ejecutar programas complicados como el siguiente:

2 Programa de ejemplo en MiniLPP

MiniLPP Sample Code

```
Entero a, b
Entero x, _y, z

// Funcion que calcula el maximo comun divisor
Funcion gcd(Entero a, Entero b): Entero
Inicio
    Si b == 0 Entonces
        Retorne a
    Sino
        Retorne gcd(b, a mod b)
Fin

Inicio
    a <- 10
    b <- 20
    x <- a
    _y <- b
    z <- gcd(x, _y)

    Escriba z
Fin
```

3 Notación a utilizar

<nts>	Significa que ***nts*** es un símbolo no terminal.
ts	Significa que ts es un símbolo terminal, o sea un Token reconocido por el analizador léxico.
'x'	Significa que la cadena x es un terminal cuyo lexema es x.
[r]	Significa cero o una ocurrencia de ***r***.
r*	Significa cero o más ocurrencias de ***r***.
r+	Significa una o más ocurrencias de ***r***.
{r}+,	Una lista separada por comas de una o más ocurrencias de ***r***.
{}	Las llaves son usadas para agrupar
|	Usado para separar alternativas
[c1-c2]	Denota un conjunto de caracteres. Ej: **[a-c]** denota los tres caracteres **a, b, c**

4 Consideraciones Léxicas

Las palabras reservadas y los identificadores son case-insensitive. Por ejemplo, si es una palabra reservada, pero también lo son Si, SI y todas sus variantes. Además, compi2 y Compi2 hacen referencia al mismo identificador. Hay dos tipos de comentarios:

- De linea: Comienzan con // y terminan con un fin de línea.
- De bloque: Comienzan con /* y terminan con */

5 Definición de Tokens

Las palabras reservadas son las siguientes:

```
entero real cadena booleano caracter arreglo de funcion procedimiento var inicio fin final si entonces
sino para mientras haga llamar repita hasta caso o y no div mod lea escriba retorne tipo es registro
archivo secuencial abrir como lectura escritura cerrar leer escribir verdadero falso
```

Los operadores y tokens de puntuación son los siguientes:

```
[ ] , :  ( ) <- + - * ^ < > == != <= >=
```

Tokens como `stringConstant` el cual define una constante cadena como `"Hola Mundo"` no aparecen en la lista anterior pero son tokens válidos y son usados en la definición de la gramática de `MiniLPP` (Vea la sección 5).

Los identificadores, denotados por el token `ID` son definidos como una cadena de caracteres que comienza con un carácter alfabético o un guión bajo, seguido de cero o más caracteres alfanuméricos incluyendo el guión bajo.

Las palabras reservadas y los identificadores deben estar separados por espacios en blanco, o un token que no sea una palabra reservada o un identificador. Ej: `enteroparaverdadero` es un solo identificador, no tres distintas palabras reservadas. Veas la sección 4.2 para algunos ejemplos.

Constantes de cadena, denotadas por el token `stringConstant` tendrán un valor léxico compuesto de caracteres encerrados entre comillas dobles. Una cadena debe comenzar y terminar en una sola línea, no puede expandirse en multiples lineas. Para más detalles sobre cadenas y caracteres vea la sección 4.3

Contantes numéricas en `MiniLPP`, denotadas por el token `intConstant`, son decimales (base 10), hexadecimales (base 16) o binarias (base 2). Una constante entera hexadecimal comienza con `0x` seguido de una secuencia de dígitos hexadecimales. Las constantes binarias comienzan con el símbolo `0b` . Ejemplos de constantes enteras:

```
8, 012, 0x0, 0x12aE, 0b01100101
```

Constantes Carácter, denotados por el token `charConstant` tendrán un valor léxico que es un solo character encerrado entre comillas sencillas. Una constante carácter es cualquier caraćter ASCII que sea imprimible (Valores ASCII enter 32 y 126). Una constante carácter no puede ser una comilla sencilla `'''`. Para más detalles sobre cadenas y caracteres vea la sección 4.3.

6 Reconocimiento de Tokens y Espacios en Blanco

El reconocimiento de tokens tales como constants enteras, palabras reservadas e identificadores son explicadas usando las siguientes reglas. En efecto estas reglas definen un algoritmo para agrupar caracteres del conjunto [a-zA-Z0-9] en tokens.

- Si la secuencia comienza con `0x`, entonces este caracter y la secuencia más larga de caracteres del conjunto `[0-9a-fA-F]` que sigue forman una constante hexadecimal. El último carácter de esa secuencia marca el fin del token.
- Si la secuencia comienza con `0b`, entonces este caracter y la secuencia más larga de caracteres del conjunto `{0, 1}` que sigue forman una constante binaria. El último carácter de esa secuencia marca el fin del token.
- Si la secuencia comienza con un dígito decimal entonces la secuencia más larga de dígitos decimales forman una constante entera. Tenga en cuenta que la semántica de verificación de rango se llevará a cabo luego, de esta forma la secuencia 123456789123456789 la cual claramente está fuera de rango, será reconocida como un solo token por el lexer.
- Si la secuencia comienza con un carácter alfabético ó `_`, entonces este caracter y la secuencia más larga de caracteres alfanuméricos `[0-9a-zA-Z_]` que siguen a este caracter inicial forman un token que puede ser un identificador o una palabra reservada.
- Espacios en blanco y otras definiciones de tokens juegan un papel importante en la delimitación de tokens. Por ejemplo la cadena `mod3` es un solo identificador, pero `mod 3` son dos tokens, la palabra reservada `mod` y el token constante entera 3, `mod(3)` representa 4 tokens, la palabra reservada `mod`, el paréntesis izquierdo, la constante entera `3` , y el paréntesis derecho. Se consideran espacios en blanco los caracteres de fin de linea, tabulador y el carácter ASCII de espacio en blanco.

Aquí hay varios ejemplos de las reglas explicadas anteriormente:

```
0x123food = INT_CONST(123f, 16), IDENTIFIER(ood)
0xfood123 = INT_CONST(f, 16), IDENTIFICADOR(ood123)
123retorne = INT_CONST(123, 10), KEYWORD(RETORNE)
0x123mod3 = INT_CONST(123, 16), IDENTIFICADOR(mod3)
0x123mod 3 = INT_CONST(123, 16), KEYWORD(MOD), INT_CONST(3, 10)
1250x356 = INT_CONST(1250, 10), IDENTIFICADOR(x356)
mientras123 = IDENTIFICADOR(mientras123)
retornefuncion = IDENTIFICADOR(retornefuncion)
```

7 Constantes de cadena y carácter

Las constantes de cadenas, denotadas por el token `stringConstant` tendrán un valor léxico compuesto de los caracteres encerrados entre comillas dobles. Una cadena debe comenzar y terminar en una sola línea, no puede dividirse en múltiples líneas. Para poder insertar una comilla doble en una literal de cadena utilizaremos doble comilla. Las constantes de carácter, denotadas por el token `charConstant` tendrán un valor léxico que es un solo carácter encerrado entre comillas sencillas. Una constante carácter es cualquier símbolo ASCII que tenga representación gráfica (Valores entre 32 y 126). Una constante carácter no puede ser una comilla sencilla ’’’ . Las constantes de carácter pueden utilizar la secuencia de escape doble comilla sencilla para definir una comilla sencilla como literal caracter. Al momento de reconocer constantes de carácter `charConstant` o constantes de cadena `stringConstant` en el analizador léxico deberá tener en cuenta lo siguiente:

- Las constantes de cadena y caracter pueden contener la secuencia de escape doble comilla
- Las constante de cadena y de carácter deberán cerrarse con una comilla doble y sencilla, respectivamente, en particular las siguientes constantes deberán tratarse como un error:
  - Constantes de caracter que contengan cero caracteres ’’
  - Constantes de cadena y de carácter sin el carácter de cierre deben ser reportadas como errores.

8 Gramática de Referencia

La siguiente gramática de referencia define la estructura de un programa de `NanoPascal`. Aquí usamos la notación definida en la sección 3. Esta gramática de referencia no es libre de contexto, aunque podría convertirse fácilmente en gramática libre de contexto.

```
   <program> -> [subtypes-section] [variable-section] <subprogram-decl>* begin [ {<statement>}+;
] [';'] end '.'
   <subtypes-section> -> {<subtype-decl>}+<EOL>
   <subtype-decl> -> tipo ID es <type>
   <variable-section> -> {<variable-decl>}+<EOL>
   <variable-decl> -> <type> {ID}+,
   <type> -> entero | booleano | caracter | <array-type>
   <array-type> -> arreglo '[' intConst '..'  intConst ']' de <type>
   <subprogram-decl> -> <subprogram-header> EOL [variable-section] inicio <statement>* fin
   <subprogram-header> -> <function-header> | <procedure-header>
   <function-header> -> funcion ID [ '(' [ <argument-decl>}+; ] ')' ] ':'  <type>
   <function-header> -> procedimiento ID [ '(' [ <argument-decl>}+, ] ')' ]
   <argument-decl> -> var <type> ID | <type> ID
   <statement> -> <lvalue> '<-' <expr>
                | llamar ID [ '(' [ {<expr>}+, ] ')' ]
                | escriba {<argument>}+,
                | lea {<lvalue>}+,
                | <if-statement>
                | mientras <expr> [EOL] haga EOL <statement>+<EOL> EOL fin mientras
                | repita EOL <statement>+<EOL> EOL hasta <expr>
                | para <lvalue> '<-' <expr> hasta <expr> haga EOL <statement>+<EOL> EOL fin para
```

```
   <if-statement> -> si <expr> [EOL] entonces <statement>+<EOL> EOL <else-if-block>* [<else-block>]
fin si
   <else-if-block> -> sino si <expr> [EOL] entonces <statement>+<EOL> EOL
   <else-block> -> else <statement>+<EOL> EOL
   <statement-block> -> <statement>
                      | inicio [ {<statement>}+<EOL> ] fin
   <argument> -> stringConstant | <expr>
   <lvalue> -> ID | ID '[' <expr> ']'
   <expr> -> <lvalue>
             | ID [ '(' [ {<expr>}+, ] ')' ]
             | <constant>
             | <expr> <bin-op> <expr>
             | '-' <expr>
             | no <expr>
             | '(' <expr> ')'
   <bin-op> -> <arith-op> | <rel-op> | <eq-op> | <cond-op>
   <arith-op> -> '+' | '-' | '*' | div | mod
   <rel-op> -> '<' | '>' | '<=' | '>='
   <eq-op> -> '=' | '!='
   <cond-op> -> y | o
   <constant> -> intConstant | charConstant | <bool-constant>
   <bool-constant> -> verdadero | falso
```

9 Reglas Semanticas

Un programa de MiniLPP consiste de declaraciones de variables y declaraciones de subprogramas a nivel global. Las variables globales pueden ser accesadas por todos los subprogramas.

9.1 Tipos de Datos

Hay tres tipos básicos en MiniLPP – Entero para enteros, Booleano para booleanos y Caracter para caracteres. Además el lenguage soporta arreglos unidimensionales de estos tipos básicos. Todos los arreglos tienen un tamaño fijo definido en tiempo de compilación.

9.2 Expresiones

Las expresiones siguen las reglas de precedencia de la tabla 1. Las constantes enteras evaluan a su valor entero. Las contantes carácter evaluan a su valor entero ASCII, ej. 'A' evalua a 65 (si tiene Linux puede ejecutar man ascii para la tabla ASCII completa). Una expresión que hace referencia a un elemento de un arreglo, ej. `x[10]` evalua al valor contenido en la posición referenciada. Los operadores relacionales son usados para comparar expresiones de tipo entero, caracter y booleano. Los operadores de igualdad `'='` y `'!='` están definidos para los tipos Entero, Caracter, y Booleano y pueden ser utilizados para comparar dos expresiones que tengan el mismo tipo. El resultado de un operador relacional o de igualdad tiene tipo Booleano. Los operadores booleanos `y` y `o` son interpretados usando corto-circuito. Esto significa que los efectos secundarios del segundo operando no son ejecutados si el resultado del primer operando determina el valor de la expresión (Esto es: si el resultado es falso para `y` o verdadero para `o`).

Operadores	Precedencia
`no, - unario`	La más alta
`* div mod y`	Segundo nivel
`+ - o`	Tercer nivel
`= != < > <= >=`	Nivel más bajo

Tabla 1: Precedencia de operadores en MiniLPP

El nível de precedencia de cada operador se muestra en la tabla anterior. Todos los operadores en el mismo nivel tienen la misma precedencia. Los operadores con igual precedencia se asocian por la izquierda.

9.3 Sentencias

Las expressiones condicionales en sentencias como `Si`, `Mientras` y `Repita` deben evaluar al tipo `Booleano`, en caso de no ser asi el compilador deberá generar un error.

Las asignaciones solo son válidas para tipos de datos `Entero`, `Caracter`, y `Booleano`, esto implica que no se pueden asignar arreglos.