El lenguaje de programación algorítmica simplificada PrAlSim

1.- Introducción

En los años que llevan en marcha las titulaciones de Informática en la UPNa, se ha venido empleando en las asignaturas básicas de programación un lenguaje de programación algorítmica que permite enseñar metodología de la programación evitando los problemas inherentes a los lenguajes concretos. Obviamente, no existe hasta hoy ningún compilador para este lenguaje. En esta práctica se pretende desarrollarlo (para un subconjunto del lenguaje).

2.- Presentación del lenguaje

Un programa PrAlSim contiene una descripción de sentencias a realizar y una descripción de los datos que manipulan dichas sentencias. Las sentencias se describen mediante instrucciones y los datos por medio de declaraciones.

Los datos se representarán por valores de variables. Las variables deben declararse antes de poder ser usadas. Una declaración asocia a la variable un tipo que define el dominio de dicha variable. Los tipos escalares predefinidos son: entero, carácter y booleano. Se pueden definir subrangos de todos ellos.

El lenguaje PrAlSim permite definir tipos compuestos. Se admiten dos tipos de composiciones: tabulares (arrays) en los que todos los elementos son del mismo tipo y se accede a un elemento por medio de un índice; y registros a cuyos campos se accede a través de un nombre.

El conjunto de instrucciones de PrAlSim es reducido. Permite asignaciones de expresiones a variables, composiciones secuenciales, alternativas e iterativas.

Un programa PrAlSim está compuesto por una cabecera, un bloque principal y un conjunto de funciones. Los parámetros de las funciones son siempre de entrada.

A continuación se va a describir tanto el léxico como la sintaxis del lenguaje. Se incluyen comentarios diversos sobre la semántica.

2.1.- El Léxico

2.1.1.- Alfabeto

Todos los tokens del lenguaje se construyen empleando el siguiente conjunto de caracteres:

- Letras Mayúsculas A..Z
- Letras minúsculas a..z
- Cifras: 0..9
- Caracteres especiales " / ( ) = [ ] + * { } , ; . : - _ > <
- Caracteres espacio en blanco, tabulador y salto de línea

El lenguaje diferencia siempre mayúsculas de minúsculas

En lo que sigue, para evitar confusiones se indicaran siempre en negrita los símbolos del alfabeto cuando aparezcan en una expresión regular.

2.1.2.- Convenciones sobre espacios:

Un programa es una secuencia de tokens. La división de esta secuencia en líneas, así como el espaciamiento o el uso de tabuladores no afecta a la semántica del programa.

2.1.3.- Tokens

Los identificadores

Se utilizan para nombres de variables, tipos, nombres de campos de registros, acciones, funciones y palabras reservadas. Su forma es:

identificador → letra letra_o_cifra*
letra_o_cifra → letra | cifra
letra → letra_mayuscula | letra_minuscula

Literales numéricos

Dado que sólo trabajamos con el tipo entero, los literales numéricos responden a las expresión regular:

literal_numerico → (+ | -)?cifra cifra*

Literales de caracteres

Son cadenas de texto entre **"**. La cadena puede contener cualquier carácter válido con una excepción: el carácter **"** sólo puede aparecer si está precedido del carácter \. Es decir:

cadena_de_caracteres → **"**(carácter_no_comilla | **"**)***"**

Comentarios

Los comentarios pueden aparecer en cualquier parte del programa, aunque tienen que aparecer forzosamente dos (la precondición y la postcondición). Comienzan con una llave de apertura y finalizan por una llave de cierre. En medio de esas llaves pues ir cualquier cadena de texto con una excepción: el carácter **}** sólo puede aparecer si está precedido por \. Es decir:

comentario → **{**(carácter_no_llave_cierre | **}**)***}**

Palabras reservadas

Para facilitar la labor, vamos a utilizar un conjunto de palabras reservadas que el usuario no puede utilizar como identificadores. Son las palabras:

accion	algoritmo	const	continuar	de	dev	div	ent	e/s	entonces	faccion
falgoritmo	falso	fconst	ffuncion	fmientras	fpara	fsi	ftipo	ftupla	funcion	fvar
hacer	hasta	mientras	mod	no	o	para	sal	si	tabla	tipo
tupla	var	verdadero	y							

Otros tokens

Obviamente, pueden existir otros tokens. Se trata, de agrupaciones de caracteres que tienen una semántica especial. Recuerde que no existe una frontera clara entre lo que es un token y lo que no. Es por ello que lo que se indica a continuación son algunos ejemplos de cosas que podrían ser tokens. La decisión de si lo son o no se deja en su mano. Recorriendo la gramática que se explica a continuación puede encontrar otros ejemplos de posibles tokens.

2.2.- La Sintaxis

A continuación se explica la sintaxis del lenguaje. Se van a dar algunas ideas de cómo sería una gramática inicial, pero debe tenerse en cuenta que no es completa. En cualquier caso, para evitar errores, siempre se indicarán en negrita los tokens (nótese que no se está dando una gramática de contexto libre, de hecho en muchos casos se han utilizado expresiones regulares al explicar las reglas para ahorrar notación).

2.2.1.- Un programa en PrAlSim

El lenguaje PrAlSim permite especificar algoritmos. Un algoritmo debe contener en su definición la declaración de todas las acciones y funciones de los que vaya a hacer uso. Se presupone que el algoritmo no es interactivo en el sentido de que la interacción con el usuario es implícita a través de las variables declaradas como de entrada salida. Se presupone que el algoritmo debe incluir su especificación, por lo que se fuerza a que existan dos comentarios que se suponen contendrán la precondición (siempre comienza por Prec) y la postcondición (siempre comienza por Pos) del algoritmo. La gramática del algoritmo es:

desc_algoritmo → **algoritmo identificador** cabecera_alg bloque_alg **falgoritmo**
cabecera_alg → decl_globales decl_a_f decl_ent_sal **comentario**
bloque_alg → bloque **comentario**
decl_globales → (declaracion_tipo | declaracion_const) decl_globales | ε
decl_a_f → (accion_d | funcion_d) decl_a_f | ε
bloque → declaraciones instrucciones
declaraciones → (declaracion_tipo | declaracion_const | declaracion_var) declaraciones | ε

Las declaraciones de un bloque son sólo válidas y visibles dentro de ese bloque mientras que las globales son visibles en todo el algoritmo. Puede verse que esta estructura obliga a que todas las acciones y funciones se declaren en una única zona de programa: después de las declaraciones y antes de las declaraciones de las variables que permiten establecer la precondición y postcondición del algoritmo (es decir, las que entenderemos que se emplean para comunicar con el usuario datos y resultados)

Declaraciones

El lenguaje define varias formas de entidades con nombre: constantes, variables, tipos, campos de tuplas, acciones o funciones. Las declaraciones asignan un nombre a dichas entidades.

Las zonas en las que se declaran variables, constantes y tipos están marcadas por tokens que indican el principio y el final de la zona de declaraciones. Así

declaracion_tipo → **tipo** lista_d_tipo **ftipo;**
declaracion_cte → **const** lista_d_cte **fconst;**
declaracion_var → **var** lista_d_var **fvar;**

El ámbito de validez de estas declaraciones difiere. Los tipos y constantes no pueden usarse hasta haber sido declarados y su nombre no puede coincidir con el de ningún otro objeto de un programa. Las variables no pueden ser usadas antes de declararse pero su rango de validez es única y exclusivamente el bloque donde son definidas.

Declaraciones de tipos

Podemos declarar tipos nuevos o asignarle un nombre nuevo a un tipo conocido:

lista_d_tipo → **identificador =** d_tipo**;** lista_d_tipo | ε
d_tipo → **tupla** lista_campos **ftupla** | **tabla[**expresion_t **..** expresion_t**] de identificador**
d_tipo → **identificador** | expresion_t **..** expresion_t | ^tipo
expresion_t → expresion | carácter
lista_campos → **identificador:** tipo**;** lista_campos | ε
tipo → tipo_escalar | **identificador**

carácter → cualquier cosa de longitud 1 entre comillas dobles

Los tipos escalares básicos son los conocidos

- Entero: la implementación deberá especificar el dominio aunque debe permitir valores positivos y negativos.
- Booleano: Admite los valores verdadero y falso (terminales de la gramática). El modo de almacenamiento de estos valores depende de las implementaciones.
- Carácter: Admiten valores enteros entre 0 y 255. Suponemos que existe una tabla ASCII por detrás, pero no entramos en cual es su forma. Obviamente, "a" es un valor entre 0 y 255.

Declaración de constantes

El lenguaje permite declarar únicamente constantes de alguno de los tipos escalares básicos. La declaración se efectúa siguiendo el siguiente esquema:

lista_d_cte → **identificador =** (literal_numerico | caracter | **verdadero** | **falso);** lista_d_cte | ε

Declaración de variables

Las variables se declaran indicando identificadores y tipos. Los tipos pueden ser un identificador o una d_tipo. Un tipo que aparezca en una declaración de variables no puede usarse más que por las variables que aparezcan en la declaración

lista_d_var → lista_id **:** (**identificador** | d_tipo)**;** lista_d_var | ε
lista_id → **identificador,** lista_id | ε

Un caso particular de declaración de variables es el de las variables de entrada salida del algoritmo. Estas se declaran por medio de la estructura sintáctica:

decl_ent_sal → decl_ent | decl_ent decl_salida | decl_salida
decl_ent → **ent** lista_d_var
decl_sal → **sal** lista_d_var

Expresiones

El lenguaje trabaja con expresiones aritméticas sencillas y expresiones lógicas. Se supone que, en las expresiones aritméticas, el compilador debe diferenciar el tipo del resultado en función de los tipos de los operandos. Los operandos pueden ser constantes, variables o partes de variables estructuradas.

expresion → exp_a | exp_b | funcion_ll
exp_a → exp_a **+** exp_a | exp_a **–** exp_a | exp_a ***** exp_a | exp_a **div** exp_a | **-** exp_a | exp_a **mod** exp_a
exp_a → **(**exp_a**)** | operando **| literal_numerico**
exp_b → exp_b **y** exp_b | exp_b **o** exp_b | **no** exp_b | operando | **verdadero** | **falso**
exp_b → expresion oprel expresion | **(**exp_b**)**

operando → **identificador** | operando**.**operando | operando**[**expresion**]** | operando^
oprel → = | < | > | <= | >= | <>

Las normas sobre conmutatividad y asociatividad de los operadores indicados son las habituales.

Instrucciones

Se definen como instrucciones la asignación la composición secuencial, la alternativa y la composición iterativa.

instrucciones → instrucción**;** instrucciones | instrucción
instrucción → **continuar** | asignacion | alternativa | iteracion | accion_ll
asignacion → operando **:=** expresion
alternativa → **si** expresion **->** instrucciones lista_opciones **fsi**
lista_opciones → **[]** expresion **->** instrucciones lista_opciones | ε
iteracion → it_cota_fija | it_cota_exp
it_cota_exp → **mientras** expresion **hacer** instrucciones **fmientras**
it_cota_fija → **para** asignacion **hasta** expresion **hacer** instrucciones **fpara**

Nótese que el terminal ; se emplea como separador de instrucciones, no como finalizador.

El compilador va a presuponer que la alternativa es exclusiva. Es decir, que la conjunción de todas las opciones es verdadero y que nunca son verdaderas dos alternativas a la vez. No se va a comprobar este hecho pues comprobar la veracidad de la conjunción es un problema NP-completo. Sin embargo, se pide que el compilador tenga en cuenta la exclusividad de la alternativa a la hora de generar código.

En la iteración acotada por expresión se evaluará la expresión y si es verdadera, se ejecutarán las instrucciones. Si es falsa se pasará a la siguiente expresión. En la iteración de cota fija la asignación debe ser simple y hacia una variable de tipo entero. Si el valor de la variable es menor o igual que el resultado de la expresión, se realizarán las instrucciones. Tras realizarse las instrucciones se asignará a la variable su sucesor.

Acciones y funciones

El lenguaje admite acciones como instrucciones y funciones como expresiones. No se puede emplear una acción ni una función no declarada previamente. La declaración consiste simplemente en el propio código de la función. Es decir, en un programa el código de acciones y funciones debe ir por delante que el programa principal. Siguen las gramáticas:

accion_d → **accion** a_cabecera bloque **faccion**
funcion_d → **funcion** f_cabecera bloque **dev** expresion **ffuncion**
a_cabecera → **identificador (** d_par_form **);**
f_cabecera → **identificador (** lista_d_var **) dev tipo;**

La expresión de funcion_d debe ser de un tipo compatible con el indicado en f_cabecera. Sólo se pueden devolver tipos básicos.

d_par_form → d_p_form**;** d_par_form | ε
d_p_form → **ent** lista_id **: tipo** | **sal** lista_id **: tipo** | **e/s** lista_id **: tipo**

Los parámetros indicados como ent son parámetros de entrada. Es decir, la acción usa los valores que se le pasen a través de ellos pero no puede modificarlos. Los de salida (sal) son parámetros a los que la acción debe asignar un valor sin utilizar su contenido en la entrada. En los de entrada salida (e/s) la acción puede usar sus valores y modificarlos. En las funciones sólo podemos tener parámetros de entrada. De hecho, el lenguaje pide que ni se indique que el parámetro es de entrada.

Las llamadas a acciones y funciones siguen el esquema

accion_ll → **identificador** (l_ll)
funcion_ll → **identificador** (l_ll)
l_ll → expresión**,** l_ll | expresion

La lista l_ll será una lista con el mismo número de parámetros que los que se dieron cuando se definió la acción/función. Los tipos deben coincidir uno a uno. A un parámetro formal de salida o de entrada salida debe corresponderle en l_ll una variable del tipo adecuado. A un parámetro de entrada puede corresponderle una variable o una expresión. En ambos casos, el tipo debe ser el adecuado.

3.- El lenguaje intermedio

Como lenguaje intermedio vamos a emplear un código de tres direcciones ampliado con dos instrucciones. Son:

- Input variable
- Output variable

Las instrucciones input variable serán el código que deba generarse cuando se lea la definición de una variable de entrada del algoritmo. Las acciones output variable serán las que se produzcan, cuando finalice la ejecución del algoritmo para todas las variables de salida del algoritmo.

4.- Tratamiento de errores

El compilador avisa al usuario de los errores cometidos acreciéndole la máxima información posible. Si es posible, no detiene su ejecución, si no que trata de resolver el error y seguir compilando.

Ejercicio Práctico

Práctica:

Cree un compilador que admita el lenguaje de programación PrAlSim y genere código en lenguaje intermedio.

El lenguaje intermedio generado debe ser previamente acordado con el profesor. Dado que no se está generando código ejecutable, será necesario también generar, en algún formato acordado, un fichero con la tabla de símbolos.

Mínimo para superar la práctica: Como mínimo deben superarse los siguientes hitos:

1. Análisis de las aplicaciones flex y bison
2. Análisis de la gramática
3. Construcción de una pareja scanner-parser que se comuniquen correctamente.
4. Debe generarse código en tres direcciones para, al menos, programas escritos para una versión reducida de PrAlSim que no contenga apuntadores, registros, acciones ni funciones

Ejemplos

Algunos Ejemplos de algoritmos que deberían compilarse son:

Algoritmo para intercambiar dos valores

```
algoritmo intercambio;
ent a, b: entero;
sal a, b: entero;
{Prec – a= A AND b = B}
var
   aux: entero;
fvar
   aux:= a;
   a:= b;
   b:= aux
{Post – b=A AND a = B}
falgoritmo
```

El código de tres direcciones que debería generarse debería ser similar a::

```
1 input a
2 input b
3 aux := a
4 a:= b
5 b := aux
6 output a
7 output b
```

Algoritmo para ordenar tres valores

```
algoritmo orden3;
ent a, b, c: entero;
sal max, min, med: entero;
{Prec – a = A AND b = B AND c = C}
   si a >= b y a >= c ->
      max := a;
      si b >= c ->
         med := b;
         min := c
```

```
      [] b < c ->
         med := c;
         min := b
      fsi
   [] b >= a y b >= c ->
      max := b;
      si a >= c ->
         med := a;
         min := c
      [] a < c ->
         med := c;
         min := a
      fsi
   [] c >= b y c >= a ->
      max := c;
      si b >= a ->
         med := b;
         min := a
      [] b < a ->
         med := a;
         min := b
      fsi
   fsi
{Post — a=A AND b = B AND c= C …}
falgoritmo
```

El código de tres direcciones que debería generarse debería ser similar a::

```
1 input a
2 input b
3 input c
4 if a >= b goto 6
5 goto 17
6 if a >= c goto 8
7 goto 17
8 max := a
9 if b >= c goto 11
10 goto 14
11 med := b
12 min := c
13 goto 16
14 med := c
15 min := b
16 goto
17 ...
```

Algoritmo de suma rápida

```
algoritmo sumacomb;
ent n, m: entero;
sal suma: entero;
{Prec: n= N AND m = M AND M >= N > 0}
var
   i, comb: entero;
fvar
   i:= 0;
   suma:= 0;
   comb:= m;
   mientras i < n hacer
      suma := suma + comb;
      comb := comb * (m - i -1) div (i + 2);
      i := i+1
   fmientras;
{Post — n =N AND suma = SUMATORIO(i=1..N}(M sobre i)}
falgoritmo
```

El código de tres direcciones que debería generarse debería ser similar a::

```
1 input m
2 input n
3 i := 0
4 suma:= 0
5 comb:= m
6 if i < n goto 8
7 goto 19
8 t1 := suma + comp
9 suma := t1;
10 t2 := m — i
11 t3 := t2 — 1
12 t4 := comb * t3
```

```
13 t5 := i + 2
14 t6 := t4 / t5
15 comb := t6
16 t7 := i +1
17 i := t7
18 goto 8
```
19 output suma

Algunos hitos intermedios

1. Obtención de documentación
    a. http://flex.sourceforge.net/manual/
    b. http://www.gnu.org/software/bison/manual/
2. Lectura de manuales
3. Construcción (comprendiendo) de ejemplos de bison
4. Definición del fichero para flex
5. Definición del fichero para bison (%left, %right, %nonasoc)
6. Comprobación/mejora de conflictos (ambigüedades)
7. Asignación de tipos (YYSTYPE, yylval, %union %type)
8. Creación de la estructura de la tabla de símbolos
9. Creación de rutinas semánticas