Universidad Nacional Autónoma de México
Facultad de Ingeniería
Compiladores
Práctica Definición Dirigida por Sintaxis
y Esquema de traducción
Profesor: Adrián Ulises Mercado Martínez

Definición Dirigida por Sintaxis y Esquema de traducción

Objetivo:

Para la gramática de la sección de gramática obtener la definición dirigida por sintaxis y su respectivo esquema de traducción.

Gramática

1. programa → declaraciones **\n** funciones
2. declaraciones → tipo lista_var **\n** declaraciones
   | **registro \n inicio** declaraciones **\n fin \n** declaraciones
   | ε
3. tipo → base tipo_arreglo
4. base → **ent** | **real** | **dreal** | **car** | **sin**
5. tipo_arreglo → **[num]** tipo_arreglo | ε
6. lista_var → lista_var **, id** | **id**
7. funciones → **func** tipo **id(** argumentos **) inicio \n** declaraciones sentencias **\n fin \n** funciones | ε
8. argumentos → listar_arg | **sin**
9. lista_arg → lista_arg arg | arg
10. arg → tipo **id**
11. sentencias → sentencias**\n** sentencia | sentencia
12. sentencia → **si** expresion_booleana **entonces \n** sentencias **\n fin**
    | **si** expresion_booleana **\n** sentencias **\n sino \n** sentencias **\n fin**
    | **mientras \n** expresion_booleana **hacer \n** sentencias **\n fin**
    | **hacer \n** sentencia **\n mientras que** expresion_booleana
    | **id :=** expresion | **escribir** expresion | **leer** variable | **devolver**
    | **devolver** expresion | **segun (** expresion **) \n** casos predeterminado **\n fin**| **terminar**
13. casos → **caso num: \n** sentencias **\n** | casos **\n caso num: \n** sentencias **\n**
14. predeterminado → **predet:\n** sentencias | ε
15. expresion_booleana → expresion_booleana **oo** expresion_booleana
    | expresion_booleana **yy** expresion_booleana
    | **no** expresion_booleana
    | relacional | **verdadero** | **falso**
16. relacional → relacional < relacional | relacional > relacional | relacional <= relacional
    | relacional >= relacional | relacional == relacional | relacional <> relacional | expresion
17. expresion → expresion + expresion | expresion **-** expresion
    | expresion ***** expresion | expresion / expresion
    | expresion % expresion | (expresion)
    | variable | **num** | **cadena** | **caracter** | **id(** parametros **)**

18. param_arr → **id**[ ] | param_arr [ ]
19. variable → **id** parte_arreglo | **id.id**
20. parte_arreglo → [expresion] parte_arreglo | ε
21. parametros → lista_param | ε
22. lista_param → lista_param **,** expresion | expresion

Ejercicios

Consideraciones semánticas

1. Los índices para los arreglos solo pueden ser de tipo entero.
2. Los arreglos al ser declarados no pueden contener números menores que cero.
3. En la llamadas a funciones se debe validar el tipo y número de argumentos.
4. Es opcional permitir la recursividad en la llamadas a funciones.
5. Se debe comprobar todos los tipos de retorno en una función que coincidan con el tipo de la función.
6. Una función no debe poder devolver tipos registro.
7. Se debe prohibir declarar variables de tipo registro y sin tipo dentro de las funciones.
8. No se puede declarar variables sin tipo en ninguna parte del programa.
9. Las variables de tipo registro serán solo globales
10. Las funciones no pueden devolver registros, solo los tipos nativos del lenguaje.
11. Sólo las funciones pueden ser sin tipo.
12. Se debe realizar conversión de tipos ya sea de ampliación o reducción según el caso.
13. Se debe validar la existencia de un identificador en la tabla de símbolos global o local, además para los identificadores de los campos de un registro se debe buscar en la tabla de símbolos que define a ese registro para ver si existe el identificador.
14. Las etiquetas no pueden ser atributos heredados por lo tanto usar backpatch.
15. El atributo código debe ser una variable global, puede ser una lista ligada donde se van agregando las cuádruplas.
16. La generación de código destino se realiza al final del proceso de compilación.
17. Considere que la instrucción terminar es equivalente a break en lenguaje C
18. Escriba sus conclusiones de forma individual

Anexo

Cuádruplas

Una cuádrupla es una forma de representar el código de tres direcciones de la siguiente forma
operador operando1 operando2 resultado
Se puede usar una estructura de datos como la que sigue:

```
typedef struct _quad quad;

struct _quad{
  char op[32];
  char arg1[32];
  char arg2[32];
  char res[32];
  quad *next;
};

typedef struct _code code;

struct _code{
  quad *root;
  int num_instrucciones;
};

quad* crea_quad(char *op, char* arg1, char *arg2, char* res);
void elimna quad(quad *q);
code *crea_code();
void elimina_code(code *c);
void agrega_cuadrupla(code* c, char *op, char* arg1, char *arg2, char* res);
```