UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE INGENIERÍA
COMPILADORES

AMALFI FIGUEROA ISAAC
ÁLVAREZ SOLÍS SERGIO IVÁN
ESCAMILLA JAIMES NEFTALÍ
TAPIA ÁLVAREZ JORGE SAÚL

21 JUNIO 2020

Tabla de contenido

OBJETIVO

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

INTRODUCCIÓN

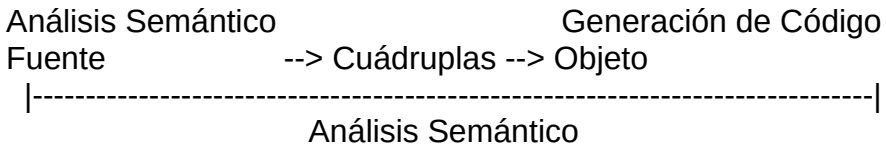
GENERACIÓN DE CUADRUPLAS

El código generado por un compilador puede ser de uno de los tipos siguientes:

- Código simbólico (*.ASM, hay que pasárselo a un ensamblador).
- Código relocalizable (*.OBJ, hay que pasárselo a un "linker"). Es el sistema más flexible y el más corriente en compiladores comerciales.
- Código absoluto (*.EXE, más eficiente pero menos flexible. Sólo en compiladores antiguos).

Existen dos formas de realizar la generación de código:

- En un solo paso: integrada con el análisis semántico.
- En dos o más pasos: el analizador semántico genera un código intermedio (cuádruplas, notación sufija), a partir del cuál se realiza la generación del código definitivo como un paso independiente.



Una cuádrupla es una estructura de tipo registro con cuatro campos: op, result, arg1 y arg2.

- Por ejemplo, la proposición de tres direcciones x = y + z se representaría como: (suma, x, y, z).
- En general, las instrucciones que no requieren todos los campos dejan vacíos los que no utilizan
    - Ej: Las proposiciones con operadores unarios no utilizan arg2

COMPROBACIÓN DE TIPOS

Son las reglas de un lenguaje que permiten asignar tipos a las distintas partes de un programa y verificar su corrección, formado por las definiciones y reglas que permiten comprobar el dominio de un identificador, y en qué contextos puede ser

usado, cada lenguaje tiene un sistema de tipos propio, aunque puede variar de una a otra implementación. La comprobación de tipos es parte del análisis semántico.

Algunas de sus funciones principales son:

- Inferencia de tipos: calcular y mantener la información sobre los tipos de datos.
- Verificación de tipo: asegurar que las partes de un programa tienen sentido según las reglas de tipo del lenguaje.

La información de tipos puede ser estática o dinámica: LISP, CAML o Smalltalk utilizan información de tipos dinámica mientras que en ADA, Pascal o C la información de tipos es estática. También puede ser una combinación de ambas formas.

Cuantas más comprobaciones puedan realizarse en la fase de compilación, menos tendrán que realizarse durante la ejecución y por lo tanto mayor eficiencia del programa objeto.

GRAMÁTICA

sin: significa sin tipo, car: tipo caracter

1. programa → declaraciones funciones
2. declaraciones → tipo lista_var; declaraciones
   | tipo_registro lista_var; declaraciones
   | ε
3. tipo_registro → **estructura inicio** declaraciones **fin**
4. tipo → base tipo_arreglo
5. base → **ent** | **real** | **dreal** | **car** | **sin**
6. tipo_arreglo → **(num)** tipo_arreglo | ε
7. lista_var → lista_var, **id** | **id**
8. funciones → **def** tipo **id(**argumentos**) inicio** declaraciones sentencias **fin** funciones
   | ε
9. argumentos → listar_arg | **sin**
10. lista_arg → lista_arg, arg | arg
11. arg → tipo_arg **id**
12. tipo_arg → base param_arr
13. param_arr → **( )** param_arr | ε
14. sentencias → sentencias sentencia | sentencia
15. sentencia → **si** e_bool **entonces** sentencia **fin**
    | **si** e_bool **entonces** sentencia **sino** sentencia **fin**
    | **mientras** e_bool **hacer** sentencia **fin**
    | **hacer** sentencia **mientras** e_bool;
    | **segun (**variable**) hacer** casos predeterminado **fin**
    | variable **:=** expresion **;**
    | **escribir** expresion **;**
    | **leer** variable **;** | **devolver;**
    | **devolver** expresion;
    | **terminar;**
    | **inicio** sentencias **fin**
16. casos → **caso num:** sentencia casos | **caso num:** sentencia
17. predeterminado → **pred:** sentencia | ε
18. e_bool → e_bool **o** e_bool | e_bool **y** e_bool | **no** e_bool
    | relacional | **verdadero** | **falso**

19. relacional → relacional > relacional
    | relacional < relacional
    | relacional <= relacional
    | relacional >= relacional
    | relacional <> relacional
    | relacional = relacional
    | expresion

20. expresion → expresion + expresion
    | expresion − expresion
    | expresion * expresion
    | expresion / expresion
    | expresion **%** expresion | **(**expresion **)** |
    | variable | **num** | **cadena** | **caracter**

21. variable → **id** variable_comp

22. variable_comp → dato_est_sim | arreglo | **(** parametros **)**

23. dato_est_sim → dato_est_sim **.id** | ε

24. arreglo → **(** expresion **)** | arreglo **(** expresion **)**

25. parametros → lista_param | ε

26. lista_param → lista_param, expresion | expresion

DESARROLLO

Análisis Léxico

La primera fase de escáner funciona como un texto escáner. Esta fase busca en el código fuente como una secuencia de caracteres y la convierte en un lexema resultante-. Analizador Léxico representa estos lexema resultante- en forma de fichas:

```
<token-name, attribute-value>
```

Sintaxis Análisis

La siguiente fase se denomina la sintaxis análisis o **análisis**. Toma el token de análisis léxico como entrada y genera un árbol analizar (o árbol de sintaxis). En esta fase, token arreglos se contrastan con el código fuente gramática, es decir, el analizador comprueba si la expresión de los tokens es sintácticamente correcto.

Análisis semántico

Análisis semántico comprueba si el análisis árbol construido sigue las reglas del idioma. Por ejemplo, la asignación de valores es entre tipos de datos compatibles, y añadiendo cadena en un número entero. Además, el analizador semántico realiza un seguimiento de los identificadores, sus tipos y expresiones; si los identificadores se declaran antes de su uso, o no, etc. El analizador semántico produce un árbol de sintaxis anotado como una salida.

Generación de código intermedio

Tras análisis semántico el compilador genera un código intermedio del código fuente para el equipo de destino. Es un programa para algunos la máquina abstracta. , Está entre el lenguaje de alto nivel y el lenguaje de máquina. Este código intermedio debe ser generado de tal manera que hace que sea más fácil de traducir en la máquina de destino.

Optimización de código

La siguiente fase de optimización de código es el código intermedio. La optimización puede ser asumida como algo que elimina código innecesario, y organiza la secuencia de declaraciones con el fin de acelerar la ejecución del programa sin desperdicio de recursos (CPU, memoria).

Generación de código

En esta fase, el generador de código optimizado la representación del código intermedio y la asigna a la máquina de destino. El generador de código se traduce el código intermedio en una secuencia de (generalmente) reubicables código máquina. Secuencia de instrucciones de código máquina realiza la tarea como el código intermedio.

Tabla de símbolos

Es una estructura de datos mantendrá en todas las fases de un compilador. Todos los nombres de identificador junto con sus tipos se almacenan aquí. La tabla de símbolos hace que sea más fácil para que el compilador pueda buscar con rapidez el registro de código y recuperarla. La tabla de símbolos se utiliza también para el campo.

CONCLUSIONES

Hemos verificado con este programa, uno de los pasos del análisis semántico, para expresiones regulares, comprobando prácticamente el funcionamiento de un compilador.

BIBLIOGRAFÍA

- http://www.cc.uah.es/ie/docencia/ProcesadoresDeLenguaje/ProcesadoresDeLenguajeTema5_1xpagina.pdf
- http://www.di-mare.com/adolfo/cursos/2008-1/pp-GenCodInterMed.pdf
- http://mmedia1.fi-b.unam.mx/SistemaDeEvaluacion/view/index.php#