Representaciones Intermedias

CI4721 – Lenguajes de Programación II

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Copyright ©2012-2016

UNIVERSIDAD SIMÓN BOLÍVAR

Representaciones intermedias

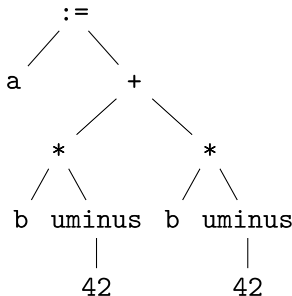
- Producto de la fase de análisis – insumo de la fase de síntesis.
  - Detalles del lenguaje origen analizados en el *front-end*.
  - Detalles del lenguaje destino sintetizados en el *back-end*.
- Simplifica agregar destinos de traducción (*retargeting*)
  - Escribir un *back-end* apropiado.
  - No es necesario cambiar el *front-end*.
  - Extremo – varios *front-end*, varios *back-end* (GNU Compiler Collection)
- Optimización de código independiente de la máquina destino.

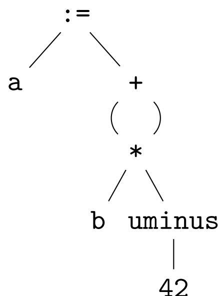
Representaciones intermedias de alto y bajo nivel pueden combinarse en un mismo compilador.

Representaciones Gráficas

Arboles Sintácticos y otras hierbas

- Arboles Sintácticos Abstractos (*Abstract Syntax Trees* o AST).
    - Representan la estructura jerárquica del programa origen.
    - Identiticar expresiones comunes lo convierte en ASD.





Construcción del Arbol Dirigida por Sintaxis

No debe ser un secreto para ustedes. . .

$S \rightarrow \textbf{id} := E$
$E \rightarrow E_1 + E_2$
$E \rightarrow E_1 * E_2$
$E \rightarrow - E_1$
$E \rightarrow (E_1)$
$E \rightarrow \textbf{id}$
$E \rightarrow \textbf{num}$

Construcción del Arbol Dirigida por Sintaxis

No debe ser un secreto para ustedes...

$$
\begin{array}{ll}
S \rightarrow \textbf{id} := E & \{\ S.tree \leftarrow mkbnode(':=', mkleaf(\textbf{id}, \textbf{id}.sym), E.tree)\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.tree \leftarrow mkbnode('+', E_1.tree, E_2.tree)\ \} \\
E \rightarrow E_1 * E_2 & \{\ E.tree \leftarrow mkbnode('*', E_1.tree, E_2.tree)\ \} \\
E \rightarrow -\ E_1 & \{\ E.tree \leftarrow mkunode('\text{uminus}', E_1.tree)\ \} \\
E \rightarrow (\ E_1\ ) & \{\ E.tree \leftarrow E_1.tree\ \} \\
E \rightarrow \textbf{id} & \{\ E.tree \leftarrow mkleaf(\textbf{id}, \textbf{id}.sym)\ \} \\
E \rightarrow \textbf{num} & \{\ E.tree \leftarrow mkleaf(\textbf{num}, \textbf{num}.val)\ \}
\end{array}
$$

- **id**.*sym* apunta a la entrada de **id** en la tabla de símbolos.
- Se obtiene un ASD si los constructores operan en modo *singleton*.

UNIVERSIDAD SIMÓN BOLÍVAR

Fácil irse por las ramas

- Facilita la reestructuración del código fuente para optimización previa.
- Requieren mucha memoria.

Preferidos para interpretadores puros
o primer paso de refinación intermedia.

UNIVERSIDAD SIMÓN BOLÍVAR

Notación Postfija

Yoda like speak must you

- Es una representación lineal del AST.
    - Lista de nodos del árbol.
    - Cada nodo aparece inmediatamente después de sus hijos.
- **Máquina Abstracta de Pila** – look ma, no registers!
    - `push v` – empilar el *valor numérico*.
    - `pop` – descartar el tope de la pila.
    - `add`, `mul`, `...` – Aritmética sobre los dos elementos al tope.
    - `rvalue l` – empilar los *contenidos* de la ubicación.
    - `lvalue l` – empilar la *dirección* de la ubicación.
    - `assign` – el *r-value* en el tope es almacenado en el *l-value* por debajo, y ambos son retirados de la pila.
    - Toda suerte de saltos incondicionales o condicionales basados en el valor al tope de la pila.

Notación Postfija Dirigida por Sintaxis

$S \rightarrow \textbf{id} := E$

$E \rightarrow E_1 + E_2$
$E \rightarrow E_1 * E_2$
$E \rightarrow - E_1$
$E \rightarrow (E_1)$
$E \rightarrow \textbf{id}$
$E \rightarrow \textbf{num}$

Notación Postfija Dirigida por Sintaxis

$$
\begin{array}{ll}
S \rightarrow \textbf{id} := E & \left\{\begin{array}{l} emit('\text{lvalue}', \textbf{id}.lexema) \\ emit('\text{assign}') \end{array}\right\} \\
E \rightarrow E_1 + E_2 & \{\ emit('\text{add}')\ \} \\
E \rightarrow E_1 * E_2 & \{\ emit('\text{mul}')\ \} \\
E \rightarrow -\ E_1 & \{\ emit('\text{uminus}')\ \} \\
E \rightarrow (\ E_1\ ) & \\
E \rightarrow \textbf{id} & \{\ emit('\text{rvalue}', \textbf{id}.lexema)\ \} \\
E \rightarrow \textbf{num} & \{\ emit('\text{push}', \textbf{num}.val)\ \}
\end{array}
$$

- Es la traducción a postfijo, pero en lugar de evaluar emite código.
- *emit* genera la cadena adecuada según los argumentos.

Notación Postfija Dirigida por Sintaxis

$$
\begin{array}{ll}
S \rightarrow \textbf{id} := E & \left\{ \begin{array}{l} emit('\text{lvalue}', \textbf{id}.lexema) \\ emit('\text{assign}') \end{array} \right\} \\
E \rightarrow E_1 + E_2 & \{\ emit('\text{add}')\ \} \\
E \rightarrow E_1 * E_2 & \{\ emit('\text{mul}')\ \} \\
E \rightarrow -\ E_1 & \{\ emit('\text{uminus}')\ \} \\
E \rightarrow (\ E_1\ ) & \\
E \rightarrow \textbf{id} & \{\ emit('\text{rvalue}', \textbf{id}.lexema)\ \} \\
E \rightarrow \textbf{num} & \{\ emit('\text{push}', \textbf{num}.val)\ \}
\end{array}
$$

```
rvalue b
push 42
uminus
mul
rvalue
b push
42 uminus
mul add
lvalue a
assign
```

- Es la traducción a postfijo, pero en lugar de evaluar emite código.
- *emit* genera la cadena adecuada según los argumentos.

UNIVERSIDAD SIMÓN BOLÍVAR

Muy bajo nivel

Barato, en todo sentido...

- La generación de código postfijo es fácil y compacta – incluso en presencia de instrucciones

$$
\begin{aligned}
S \rightarrow \textbf{if } E & \quad \left\{ \begin{array}{l} out \leftarrow newlabel() \\ emit('\text{gofalse}', out) \end{array} \right\} \\
\textbf{then } S_1 & \quad \{\ emit('label', out)\ \}
\end{aligned}
$$

  - Fácil escribir un interpretador, fácil escribir un compilador.
  - Mínimo requerimiento sobre la máquina real.

Muy bajo nivel

Barato, en todo sentido...

- La generación de código postfijo es fácil y compacta – incluso en presencia de instrucciones

$$\begin{array}{lr} S \rightarrow \textbf{if } E & \left\{ \begin{array}{l} out \leftarrow newlabel() \\ emit('\text{gofalse}', out) \end{array} \right\} \\ \textbf{then } S_1 & \{ \; emit('label', out) \; \} \end{array}$$

  - Fácil escribir un interpretador, fácil escribir un compilador.
  - Mínimo requerimiento sobre la máquina real.
- Optimizar operaciones sobre una pila resulta notablemente complejo.
  - Más operaciones sobre memoria que sobre registros – muy caro.
  - Factorizar expresiones comunes es muy complicado – cuando se puede.
  - Orden de instrucciones rígido – no es trivial mover bloques.
  - Ejecuta más instrucciones que una máquina de registros equivalente.

Muy bajo nivel

Barato, en todo sentido...

- La generación de código postfijo es fácil y compacta – incluso en presencia de instrucciones

$$
\begin{array}{ll}
S \rightarrow \textbf{if } E & \left\{ \begin{array}{l} out \leftarrow newlabel() \\ emit('\text{gofalse}', out) \end{array} \right\} \\
\textbf{then } S_1 & \{\ emit('label', out)\ \}
\end{array}
$$

  - Fácil escribir un interpretador, fácil escribir un compilador.
  - Mínimo requerimiento sobre la máquina real.
- Optimizar operaciones sobre una pila resulta notablemente complejo.
  - Más operaciones sobre memoria que sobre registros – muy caro.
  - Factorizar expresiones comunes es muy complicado – cuando se puede.
  - Orden de instrucciones rígido – no es trivial mover bloques.
  - Ejecuta más instrucciones que una máquina de registros equivalente.

Método escogido por algunas máquinas virtuales, en particular la Máquina Virtual Java

Código de Tres Direcciones

Welcome to the machine...

```
x := y op z
```

- `x`, `y` y `z` pueden ser:
  - Nombres – identificadores en el programa fuente.
  - Constantes – presentes en el programa o calculadas por el compilador.
  - Temporales – nombres generados por el compilador para descomponer las expresiones complejas y facilitar su reorganización.
- `op` es alguno de los operadores aritméticos o booleanos – aridad conveniente (generalmente unarios y binarios)

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Welcome to the machine...

```
x := y op z
```

- `x`, `y` y `z` pueden ser:
  - Nombres – identificadores en el programa fuente.
  - Constantes – presentes en el programa o calculadas por el compilador.
  - Temporales – nombres generados por el compilador para descomponer las expresiones complejas y facilitar su reorganización.
- `op` es alguno de los operadores aritméticos o booleanos – aridad conveniente (generalmente unarios y binarios)

Selección de operadores es un balance entre alto nivel y bajo nivel

Tala y poda

- Representan linealmente el árbol sintáctico.
- Variables y constantes pueden aparecer en el código intermedio – desaparecen las hojas.
- Operadores y temporales corresponden a los nodos internos – desaparecen las aristas.

```
t1  :=  -  c
t2  :=  b  *  t1
t3  :=  -  c
t4  :=  b  *  t3
t5  :=  t2  +  t4
a   :=  t5
```

```
t1  :=  -  c
t2  :=  b  *  t1
t5  :=  t2  +  t2
a   :=  t5
```

UNIVERSIDAD SIMÓN BOLÍVAR

Tala y poda

- Representan linealmente el árbol sintáctico.
- Variables y constantes pueden aparecer en el código intermedio – desaparecen las hojas.
- Operadores y temporales corresponden a los nodos internos – desaparecen las aristas.

```
t1  :=  -  c
t2  :=  b  *  t1
t3  :=  -  c
t4  :=  b  *  t3
t5  :=  t2  +  t4
a   :=  t5
```

```
t1  :=  -  c
t2  :=  b  *  t1
t5  :=  t2  +  t2
a   :=  t5
```

Pensaremos en el código de tres direcciones como un "ensamblador ideal".

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Operaciones de Asignación

- Asignaciones de tres direcciones.

```
x := y op z
```

  - `op` es un operador binario aritmético o booleano.
  - `x`, `y` y `z` son nombres.

Código de Tres Direcciones

Operaciones de Asignación

- Asignaciones de tres direcciones.

```
x := y op z
```

  - op es un operador binario aritmético o booleano.
  - x, y y z son nombres.

- Asignaciones de dos direcciones.

```
x := op y
```

  - op es un operador unario aritmético o booleano.
  - op es un operador de conversión de tipos.

Código de Tres Direcciones

Operaciones de Asignación

- Asignaciones de tres direcciones.

```
x := y op z
```

  - op es un operador binario aritmético o booleano.
  - x, y y z son nombres.

- Asignaciones de dos direcciones.

```
x := op y
```

  - op es un operador unario aritmético o booleano.
  - op es un operador de conversión de tipos.

- Instrucciones de copia.

```
x := y
```

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Alteración del Flujo de Control

- Saltos incondicionales.

```
goto L
```

  - `L` es una etiqueta generada por el compilador.
  - Pasa a ejecutar la instrucción con etiqueta `L`.

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Alteración del Flujo de Control

- Saltos incondicionales.

```
goto L
```

  - `L` es una etiqueta generada por el compilador.
  - Pasa a ejecutar la instrucción con etiqueta `L`.
- Saltos condicionales.

```
if x rel y goto L
```

  - `L` es una etiqueta generada por el compilador.
  - `rel` es algún operador relacional (<, ==, >=, ...).
  - Si la relación se cumple, pasa a ejecutar la instrucción con etiqueta `L`; en caso contrario, continuará a la instrucción siguiente al `if`.

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Arreglos de posiciones de memoria

```
x := y[i]
```

- Copia en `x` el valor en la posición `i` unidades más allá de `y`.

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Arreglos de posiciones de memoria

```
x := y[i]
```

- Copia en `x` el valor en la posición `i` unidades más allá de `y`.

```
x[i] := y
```

- Copia en la posición `i` unidades más allá de `x` el valor de `y`.

La unidad de memoria suele ser un byte,
pero podría ser cualquiera conveniente.

Código de Tres Direcciones

Obtención de Apuntadores

```
x := &y
```

- Almacena en `x` la dirección de `y`.
- `y` tiene que ser un identificador o un temporal que denota un *l-value*.

El *l-value* de `y` se convierte en el *r-value* de `x`

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Aplicación de Apuntadores

```
x := *y
```

- Almacena en `x` el *r-value* apuntado por `y`.
- `y` tiene que ser un identificador o temporal cuyo *r-value* corresponde a una dirección de memoria.

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Aplicación de Apuntadores

```
x := *y
```

- Almacena en `x` el *r-value* apuntado por `y`.
- `y` tiene que ser un identificador o temporal cuyo *r-value* corresponde a una dirección de memoria.

```
*x := y
```

- Almacena en el *l-value* apuntado por `x` el *r-value* de `y`.
- `x` tiene que ser un identificador o temporal cuyo *r-value* corresponde a una dirección de memoria.

UNIVERSIDAD SIMÓN BOLÍVAR

Código de Tres Direcciones

Llamadas a Procedimientos

param x_1
param x_2
...
param x_n
call p,n

- Equivalente a la llamada $p(x_1, x_2, \ldots, x_n)$.
- x_i es un identificador, valor constante o temporal.
- `r := call p,n` – si el procedimiento retorna valores.
- n no es redundante – algunos de los parámetros podrían utilizarse nuevamente en llamadas que suceden al `call`.
- `return r` – retornar de un procedimiento, con el valor `r` opcional.

UNIVERSIDAD SIMÓN BOLÍVAR

Etiquetas para saltos

¿Algunas o todas?

```
do
  i = i + 1;
while (x[i] < v);
```

- `i` es **int** – 4 bytes
- `x` es arreglo de **float** – 8 bytes cada uno

Etiquetas para saltos

¿Algunas o todas?

```
do
  i = i + 1;
while (x[i] < v);
```

- `i` es **int** – 4 bytes
- `x` es arreglo de **float** – 8 bytes cada uno

```
L: t1 := i + 1
   i  := t1
   t2 := i * 8
   t3 := x [ t2 ]
   if t3 < v goto L
```

```
100: t1 := i + 1
101: i  := t1
102: t2 := i * 8
103: t3 := x [ t2 ]
104: if t3 < v goto 100
```

Depende de la implantación – cualquiera de las dos está bien.

Implantación de Código de Tres Direcciones

- El código generado tendrá que ser manipulado:
  - Por el generador de código intermedio – para el flujo de control.
  - Por el generador de código final.
  - Por ambos optimizadores de código.
- Representaciones basadas en registros del lenguaje anfitrión.
  - Campos del registro representan componentes de cada instrucción.
  - Aprovechar la tabla de símbolos para ahorrar espacio.

UNIVERSIDAD SIMÓN BOLÍVAR

Cuádruples o *Quads*

- Registro con cuatro campos:
  - *op* para el operador
  - *arg1* y *arg2* para los argumentos
  - *result* para el resultado
- Omisión de campos según el caso.
  - Operadores unarios omiten *arg2*.
  - Operadores como `param` o `return` omiten *arg2* y *result*.
- Interpretación alternativa para campos según el caso.
  - Saltos usan *result* para almacenar la etiqueta.
  - Asignaciones usan *result* para almacenar el lado izquierdo.
- Los campos *op*, *arg1*, *arg2* y *result* apuntan a la tabla de símbolos cuando contienen nombres.

UNIVERSIDAD SIMÓN BOLÍVAR

Cuádruples o *Quads*

Un ejemplo simple

```
a := b * -c + b * -c
```

	op	arg_1	arg_2	*result*
0	uminus	c		t1
1	*	b	t1	t2
2	uminus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	:=	t5		a
⋮	⋮	⋮	⋮	⋮

- Facilita el reordenamiento del código durante la optimización.
- Obliga a mantener *todos* los temporales en la tabla de símbolos.

Tripletas

Remove all the temporaries!

- Reemplazar temporales por referencias a la *instrucción* que lo calculó.
- Registro con tres campos:
  - *op* para el operador.
  - *arg1* y *arg2* para los argumentos.
- Los argumentos son apuntadores.
  - A la tabla de símbolos – identificadores del programa.
  - A la tripleta que generó el valor necesario.
- Recodificar la asignación – *arg1* corresponde al *l-value*.

UNIVERSIDAD SIMÓN BOLÍVAR

Tripletas

El mismo ejemplo simple

```
a := b * -c + b * -c
```

	op	arg_1	arg_2
0	uminus	c	
1	*	b	(0)
2	uminus	c	
3	*	b	(2)
4	+	(1)	(3)
5	:=	a	(4)
⋮	⋮	⋮	⋮

- Ahorro de espacio en la tabla de símbolos y registros.
- Cantidad similar de indirecciones que al usar *quads*.
- Difícil reordenar el código durante la optimización.

UNIVERSIDAD SIMÓN BOLÍVAR

Tripletas

El acceso a los arreglos se complica

`x[i] := y`

	op	arg_1	arg_2
0	`[]=`	`x`	`i`
1	`:=`	(0)	`y`
⋮	⋮	⋮	⋮

`x := y[i]`

	op	arg_1	arg_2
0	`[]=`	`y`	`i`
1	`:=`	`x`	(0)
⋮	⋮	⋮	⋮

Requieren *dos* entradas,
que deben *mantenerse* contiguas

UNIVERSIDAD SIMÓN BOLÍVAR

Tripletas Indirectas

These are the triples you're looking for

- Tripletas usan indirección para eliminar temporales de los *Quads*.
- Tripletas Indirectas usan indirección para eliminar dependencia posicional de las Tripletas.
  - Tabla de Tripletas – tripletas construidas de la manera convencional.
  - Tabla de Instrucciones – apuntadores a la tabla de tripletas.

UNIVERSIDAD SIMÓN BOLÍVAR

Tripletas Indirectas

El mismo ejemplo simple

```
a := b * -c + b * -c
```

	triple
42	(0)
43	(1)
44	(2)
45	(3)
46	(4)
47	(5)
⋮	⋮

	op	arg_1	arg_2
0	`uminus`	`c`	
1	`*`	`b`	(0)
2	`uminus`	`c`	
3	`*`	`b`	(2)
4	`+`	(1)	(3)
5	`:=`	`a`	(4)
⋮	⋮	⋮	⋮

- Optimizador puede reordenar la Tabla de Tripletas dejando intactas las Tripletas – vuelve a ser fácil reordenar código.
- Más espacio que las Tripletas, pero menos que los *Quads*.

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de Código Intermedio

Infraestructura

- Utilizaremos Esquemas de Traducción S-Atribuidos para Generar Código Intermedio de Tres Direcciones.
- `newtemp()` – cada invocación retorna un identificador temporal único.
- `gen()` – permite construir la instrucción de tres direcciones a emitir.
    - Argumentos entre comillas simples se toman *verbatim*.
    - Argumentos que sean expresiones o atributos, se evalúan.
    - Se concatenan y emiten.
- Para cada símbolo gramatical X que sintetize código
    - $X.addr$ – ubicación para almacenar el valor (identificador o temporal).
    - $X.code$ – secuencia de instrucciones de tres direcciones asociada.

Esquema Simple Dirigido por Sintaxis

Una cadena larga...

$S \rightarrow \textbf{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow \textbf{id}$

$E \rightarrow \textbf{num}$

Esquema Simple Dirigido por Sintaxis

Un identificador sólo propaga su dirección

$S \rightarrow \textbf{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow \textbf{id}$ $\left\{ \begin{array}{l} E.addr \leftarrow \textbf{id}.addr \\ E.code \leftarrow '' \end{array} \right\}$

$E \rightarrow \textbf{num}$

- Atributo *code* acumula en una cadena – comienza vacía.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema Simple Dirigido por Sintaxis

Un temporal por cada número literal

$S \rightarrow \textbf{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow \textbf{id}$ $\quad \left\{ \begin{array}{l} E.addr \leftarrow \textbf{id}.addr \\ E.code \leftarrow '' \end{array} \right\}$

$E \rightarrow \textbf{num}$ $\quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow gen(E.addr \; ':=' \; \textbf{num}.val) \end{array} \right\}$

- Atributo *code* acumula en una cadena – comienza vacía.

Esquema Simple Dirigido por Sintaxis

Los paréntesis no cambian nada

$S \rightarrow \textbf{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1) \quad \left\{ \begin{array}{l} E.addr \leftarrow E_1.addr \\ E.code \leftarrow E_1.code \end{array} \right\}$

$E \rightarrow \textbf{id} \quad \left\{ \begin{array}{l} E.addr \leftarrow \textbf{id}.addr \\ E.code \leftarrow '' \end{array} \right\}$

$E \rightarrow \textbf{num} \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow gen(E.addr\ ':='\ \textbf{num}.val) \end{array} \right\}$

- Atributo *code* acumula en una cadena – comienza vacía.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema Simple Dirigido por Sintaxis

Un temporal para el resultado de cada operación unaria

$S \rightarrow \textbf{id} := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1 \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow E_1.code \mathrel{+\!+} \\ \qquad\qquad gen(E.addr\ ':='\ '\text{uminus}'\ E_1.addr) \end{array} \right\}$

$E \rightarrow (\ E_1\ ) \quad \left\{ \begin{array}{l} E.addr \leftarrow E_1.addr \\ E.code \leftarrow E_1.code \end{array} \right\}$

$E \rightarrow \textbf{id} \quad \left\{ \begin{array}{l} E.addr \leftarrow \textbf{id}.addr \\ E.code \leftarrow '' \end{array} \right\}$

$E \rightarrow \textbf{num} \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow gen(E.addr\ ':='\ \textbf{num}.val) \end{array} \right\}$

- Atributo *code* acumula en una cadena – comienza vacía.

Esquema Simple Dirigido por Sintaxis

Un temporal para el resultado de cada operación binaria

$$S \rightarrow \textbf{id} := E$$

$$E \rightarrow E_1 + E_2 \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow E_1.code \text{ ++ } E_2.code \text{ ++ } \\ \qquad\qquad gen(E.addr\ ':='\ E_1.addr\ '+'\ E_2.addr) \end{array} \right\}$$

$$E \rightarrow -\ E_1 \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow E_1.code \text{ ++ } \\ \qquad\qquad gen(E.addr\ ':='\ '\text{uminus}'\ E_1.addr) \end{array} \right\}$$

$$E \rightarrow (\ E_1\ ) \quad \left\{ \begin{array}{l} E.addr \leftarrow E_1.addr \\ E.code \leftarrow E_1.code \end{array} \right\}$$

$$E \rightarrow \textbf{id} \quad \left\{ \begin{array}{l} E.addr \leftarrow \textbf{id}.addr \\ E.code \leftarrow '' \end{array} \right\}$$

$$E \rightarrow \textbf{num} \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow gen(E.addr\ ':='\ \textbf{num}.val) \end{array} \right\}$$

- Atributo *code* acumula en una cadena – comienza vacía.

Esquema Simple Dirigido por Sintaxis

La asignación sólo genera código

$$S \rightarrow \textbf{id} := E \qquad \{\ S.code \leftarrow E.code \text{ ++ } gen(\textbf{id}.addr \ ':=' \ E.addr)\ \}$$

$$E \rightarrow E_1 + E_2 \qquad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow E_1.code \text{ ++ } E_2.code \text{ ++ } \\ \qquad\qquad gen(E.addr \ ':=' \ E_1.addr \ '+' \ E_2.addr) \end{array} \right\}$$

$$E \rightarrow -\ E_1 \qquad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow E_1.code \text{ ++ } \\ \qquad\qquad gen(E.addr \ ':=' \ '\text{uminus}' \ E_1.addr) \end{array} \right\}$$

$$E \rightarrow (\ E_1\ ) \qquad \left\{ \begin{array}{l} E.addr \leftarrow E_1.addr \\ E.code \leftarrow E_1.code \end{array} \right\}$$

$$E \rightarrow \textbf{id} \qquad \left\{ \begin{array}{l} E.addr \leftarrow \textbf{id}.addr \\ E.code \leftarrow '' \end{array} \right\}$$

$$E \rightarrow \textbf{num} \qquad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ E.code \leftarrow gen(E.addr \ ':=' \ \textbf{num}.val) \end{array} \right\}$$

- Atributo *code* acumula en una cadena – comienza vacía.
- La manipulación de cadenas es prohibitiva en espacio y tiempo.

Esquema Incremental Dirigido por Sintaxis

$$
\begin{array}{ll}
S \rightarrow \textbf{id} := E & \{\ gen(\textbf{id}.addr\ ':='\ E.addr)\ \} \\
E \rightarrow E_1 + E_2 & \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ gen(E.addr\ ':='\ E_1.addr\ '+'\ E_2.addr) \end{array} \right\} \\
E \rightarrow -\ E_1 & \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ gen(E.addr\ ':='\ '\mathrm{uminus}'\ E_1.addr) \end{array} \right\} \\
E \rightarrow (\ E_1\ ) & \{ E.addr \leftarrow E_1.addr\ \} \\
E \rightarrow \textbf{id} & \{ E.addr \leftarrow \textbf{id}.addr\ \} \\
E \rightarrow \textbf{num} & \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ gen(E.addr\ ':='\ \textbf{num}.val) \end{array} \right\}
\end{array}
$$

- `gen()` construye la instrucción de tres direcciones y la *agrega* a la secuencia acumulada.
  - Estructura de datos global.
  - ¡Archivo temporal!

¿Y los bloques anidados?

- Hasta ahora, los esquemas de generación han operado bajo la suposición de que todos los símbolos son globales – vamos a extenderla para manejar las situaciones típicas de anidamiento:
  - Bloques de alcance anidado – identificador "más cercano".
  - Estructuras y uniones – existencia del campo referenciado.
- El componente de análisis estático y verificación de tipos
  - Construye la tablas de símbolos jerárquica – supongamos que `lookup()` busca a través de la jerarquía.
  - Calcula los desplazamientos relativos a las bases – la tabla de símbolos contiene el atributo *offset* para cada uno.

Solamente tenemos que cambiar la generación para las reglas que acceden a identificadores

Esquema Incremental Dirigido por Sintaxis

¡Ahora manejando alcance!

$S \rightarrow \textbf{id} := E$

$E \rightarrow \textbf{id}$

Esquema Incremental Dirigido por Sintaxis

¡Ahora manejando alcance!

$S \rightarrow \textbf{id} := E$

$$E \rightarrow \textbf{id} \qquad \left\{ \begin{array}{l} p \leftarrow lookup(top, \textbf{id}.lexema) \\ \textbf{if } \text{p} = \text{nil } \textbf{then } \text{error}() \\ \textbf{else if } \text{p.level} = 0 \\ \textbf{then } E.addr \leftarrow p.addr \\ \textbf{else } E.addr \leftarrow base[p.offset] \end{array} \right\}$$

- `lookup()` busca desde el tope de la jerarquía de bloques.
  - `nil` indica que el símbolo no existe.
  - *level* indica el anidamiento – 0 es global, $>$0 es local.
  - `base` es un temporal que apuntará al registro de activación.

Esquema Incremental Dirigido por Sintaxis

¡Ahora manejando alcance!

$$S \rightarrow \textbf{id} := E \qquad \left\{ \begin{array}{l} p \leftarrow lookup(top, \textbf{id}.lexema) \\ \textbf{if } \mathrm{p} = \mathrm{nil} \textbf{ then } \mathrm{error()} \\ \textbf{else if } \mathrm{p.level} = 0 \\ \textbf{then } gen(p.addr\ ':='\ E.addr) \\ \textbf{else } gen(base[p.offset]\ ':='\ E.addr) \end{array} \right\}$$

$$E \rightarrow \textbf{id} \qquad \left\{ \begin{array}{l} p \leftarrow lookup(top, \textbf{id}.lexema) \\ \textbf{if } \mathrm{p} = \mathrm{nil} \textbf{ then } \mathrm{error()} \\ \textbf{else if } \mathrm{p.level} = 0 \\ \textbf{then } E.addr \leftarrow p.addr \\ \textbf{else } E.addr \leftarrow base[p.offset] \end{array} \right\}$$

- `lookup()` busca desde el tope de la jerarquía de bloques.
  - `nil` indica que el símbolo no existe.
  - *level* indica el anidamiento – 0 es global, >0 es local.
  - base es un temporal que apuntará al registro de activación.
- La asignación es simétrica.

UNIVERSIDAD SIMÓN BOLÍVAR

Esquema Incremental Dirigido por Sintaxis

El acceso a campos de registros requiere búsquedas en cadena

$E \rightarrow \mathbf{id}_1.\mathbf{id}_2$

Esquema Incremental Dirigido por Sintaxis

El acceso a campos de registros requiere búsquedas en cadena

$$E \rightarrow \mathbf{id}_1.\mathbf{id}_2 \quad \left\{ \begin{array}{l} r \leftarrow lookup(top, \mathbf{id}_1.lexema) \\ \textbf{if } \text{r} = \text{nil } \textbf{or } \text{r.type != record } \textbf{then } \text{error()} \\ \textbf{else} \\ \;\; f \leftarrow lookup(r.table, \mathbf{id}_2.lexema) \\ \;\; \textbf{if } \text{r} = \text{nil } \textbf{then } \text{error()} \\ \;\; \textbf{else if } \text{r.level} = 0 \\ \;\; \textbf{then } E.addr \leftarrow r.addr[f.offset] \\ \;\; \textbf{else } E.addr \leftarrow base[r.offset + f.offset] \end{array} \right\}$$

- `lookup()` busca el registro desde el tope de la jerarquía
- Si en efecto es un registro, busca en su tabla de símbolos particular.
- Siempre se usa el *offset* del campo dentro del registro.
    - A partir de la ubicación del registro cuando es global.
    - Agregado al *offset* del registro cuando es local.

Bibliografía

- [*Aho*] (Primera Edición)
  - Secciones 8.1 y 8.3
  - Ejercicios 8.1 a 8.4 y 8.6
- [*Aho*] (Segunda Edición)
  - Secciones 6.1, 6.2, 6.4.1 y 6.4.2
  - Ejercicios 6.1.1, 6.1.2 y 6.2.1
- Extienda el generador de código para incluir
  - Toma de direcciones y acceso a través de apuntadores.
  - Instrucciones para conversión implícita.
- Stack Machine – WikiPedia
- Parrot
- GNU Compiler Collection tiene RTL entre sus formas intermedias.

UNIVERSIDAD SIMÓN BOLÍVAR