Miguel Estevez

20170200

6.14

Show that, given the attribute grammar,

Grammar Rule	Semantic Rules
$decl \rightarrow type\ var-list$	$var-list.dtype = type.dtype$
$type \rightarrow \mathbf{int}$	$type.dtype = integer$
$type \rightarrow \mathbf{float}$	$type.dtype = real$
$var-list \rightarrow \mathbf{id}\ \mathbf{,}\ var-list_2$	$\mathbf{id}.dtype = var-list_1.dtype$
	$var-list_2.dtype = var-list_1.dtype$
$var-list \rightarrow \mathbf{id}$	$\mathbf{id}.dtype = var-list.dtype$

if the attribute type.dtype is kept on the value stack during an LR parse, then this value cannot be found at a fixed position in the stack when reductions by the rule $var-list \rightarrow \mathbf{id}\ occur$.

Al introducirse en la pila valores cada uno de los elemento que se reducen a var-list la posicion de la pila nunca sera estable. A menos que se retiren las reglas de copiado en la pila la posicion de dtype en la pila no sera fija.

6.20

Consider the following (ambiguous) grammar of expressions:

$$exp \rightarrow exp\ +\ exp \mid exp\ -\ exp \mid exp\ *\ exp \mid exp\ /\ exp \mid (exp) \mid \mathbf{num} \mid \mathbf{num}.\mathbf{num}$$

Suppose that the rules of C are followed in computing the value of any such expression: if two subexpressions are of mixed type, then the integer subexpression is converted to floating point, and the floating-point operator is applied. Write an attribute grammar that will convert such expressions into expressions that are legal in Modula-2: conversions from integer to floating point are expressed by applying the **FLOAT** function, and the division operator **/** is considered to be **div** if both its operands are integer.

```
exp -> exp + exp {
    if($1.type == float || $3.type == float) {
        FLOAT($1);
        FLOAT($3);
    }
    $$.type = $1.type;
    $$.val = $1 + $3
}
```

```
exp -> exp - exp {
    if($1.type == float || $3.type == float) {
        FLOAT($1);
        FLOAT($3);
    }
    $$.type = $1.type;
    $$.val = $1 - $3
}
exp -> exp * exp {
    if($1.type == float || $3.type == float) {
        FLOAT($1);
        FLOAT($3);
    }
    $$.type == $1.type;
    $$.val = $1 * $3
}
exp -> exp / exp {
    if($1.type == float || $3.type == float) {
        FLOAT($1);
        FLOAT($3);
        $$.val = $1 / $3
    }
    else{
        $$.val = div($1,$3)
    }
    $$.type == $1.type;
}
exp -> (exp){
    $$.type = $2.type;
}
exp -> num {
    $$.type = int
    $$.val = $1.val
}
exp -> num.num {
    $$.type = float
    $$.val =
}
```

6.21

Consider the following extension of the grammar of Figure 6.22(page 329) to include function declaration and calls:

program → *var-decls* **;** *fun-decls* **;** *stmts*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id** **:** *type-exp*
type-exp → **int** | **bool** | **array** **[num]** **of** *type-exp*
fun-decls → **fun** **id** **(** *var-decls* **):** *type-exp* **;** *body*
body → *exp*
stmts → *stmts* **;** *stmt* | *stmt*
stmt → **if** *exp* **then** *stmt* | **id** **:=** *exp*
exp → *exp* **+** *exp* | *exp* **or** *exp* | *exp* **[** *exp* **]** | **id** **(** *exps* **)**
| **num** | **true** | **false** | **id**
exps → *exps* **,** *exp* | *exp*

a. Devise a suitable tree structure for the new function type structures, and write a *typeEqual* function for two function types.

```
struct functionType
{
    vector<Type> variables;
    string name;
    Type ret_val;
}

bool typeEqual(functionType f1, functionType f2){
    if(f1.variables.size() != f2.variables.size() || f1.ret_val != f2.ret_vaal){
        return false;
    }
    for(int i = 0;i < f1.variables.size();i++){
        if(f1.variables[i] != f2.variables[i]){
            return false;
        }
    }
    return true;
}
```

b. Write semantic rules for the type checking of function declarations and function calls (represented by the rule $exp \rightarrow id(exps)$) similar to the rules of Table 6.10 (page 330).

```
Fun-decls -> fun id(var-decls): type-exp; cuerpo {
    $2.ret_val = $7.ret_val;
    $2.vars = $4.vars;
    $2.vars.push_back($4.ret_val);
    Insert($2.name, Type::function);
}
Exp -> id(exps){
    if(checkType($1.name) == Type::function && validateFunction($1,$3))
    {
        $$.ret_val = $1.ret_val;
    }
    else
    {
        typeError();
    }
}
```

6.22

Consider the following ambiguity in C expressions. Given the expression

(A)-x

if **x** is an integer variable and **A** is defined in a **typedef** as equivalent to **double**, then this expression casts the value of **-x** to **double**. On the other hand, if **A** is an integer variable, then this computes the integer difference of the two variables.

a. Describe how the parser might use the symbol table to disambiguate these two interpretations.

Para eliminar esta ambigüedad en sintaxis se buscaria en la tabla de simbolos cual seria el tipo de A, si A es una variable que se encuentra en la tabla de simbolos se procede con la operaction de resta, mientras que si fue definida con un **typedef** como un double entonces se procedera con un casting, esta modificacion seria hecha en la misma regla de la gramatica para cada operacion, donde la sematinca estaria dependiendo del este chequeo.

b. Describe how the scanner might use the symbol table to disambiguate these two interpretations.

No creo que el lexer sea capaz de resolver esta ambiguedad por el hecho de que el lexer desconoce los tipos de cada varibales y las operaciones que se han o se estan realizado. Haciendo que estos problemas este fuera del alcance del lexer.