FILS

Course: Compiler Techniques

Homework #6

Syntax-Directed Translation

A *syntax-directed definition* (SDD) is an attributed context-free grammar, that is a context-free grammar together with its attributes and semantic rules. (see the homework #1 on attribute grammars)

A Syntax-directed translation (SDT) is a method of translating a string into a sequence of actions by attaching one such action to each production of a grammar. SDT provides a simple way to attach semantics to the syntax definition , a Chomsky grammar in our case. Therefore, parsing a string of the grammar produces a sequence of action applications. A grammar specification embedded with actions to be performed is called a *syntax-directed translation scheme* or simply *translation schema*.

Thus, given actions and attributes of an enriched (attributed) grammar, the grammar can be used for translating strings from its language into string from another language or into some values in another than textual, semantic domain by applying the actions and carrying information through each symbol's attribute.

Exercise 1. Syntax-Directed Translation in bottom-up parsing

An SSD is *S-attributed* if every attribute is synthesized.

Let define a **Postfix Translation Scheme** (PTS) as a Syntax-Directed Translation composed from a *S-attributed, bottom-up parsable grammar* and some *semantic rules* that are placed at the end of each production of the grammar and are executed along with the reduction action of the right-side part to the left-side non-terminal of that production.

Here is an example of PTS based on a grammar for arithmetic expressions having digits as operands and **+** and ***** and parenthesis as operators:

```
L -> E n    {print(E.val);)
E -> E1 + T {E.val= E1.val+ T.val}
E -> T      { E.val= T.val; }
T -> T1 * F { T.val = T1.val x F.val; )
T -> F      { T.val = F.val; }
F -> ( E )  {F.val=E.val;)
F -> digit  { F.val = digit.lexval; )
```

where **n** is an endmarker of the expression.

The PTS defines the semantics of evaluation of arithmetic expressions. For this it uses attributed symbols of the base grammar where *val* is a synthesized attribute for the non-terminals E, T, F and *lexval* is a synthesized attribute for the terminal **digit** . Each production has an associated semantic rule that computes values of the synthesized attribute occurrences of the left-hand side non-terminal.

Use the previous example to guide your implementation of a PTS which is based on the following bottom-up parsable grammar of a block in a programming language with variable type declarations that precede the instructions (here only one command c):

```
B -> begin L ; S end
L -> L ; D
L -> D
D -> D , id
D -> real id
D -> int id
S -> c
```

The PTS should define the semantic actions that involve the symbol table in order to create new entries for each declared variable.

For the following terminal string:

begin real alfa**,** beta **; integer** gamma **; real** delta **; c end**

the symbol table should have the following structure:

Entry address	Variable name	Type	Location (offset)	. . . .
…				
101	alfa	**‘real’**	0	
102	beta	**‘real’**	8	
103	gamma	**‘integer’**	16	
104	delta	**‘real’**	20	

For this here is what you should do:

1. attribute the grammar symbols with synthesized and inherited attributes. The attributes should capture the types of variables and the offset of the current available location in the zone allocated to the variables in the block.
2. associate to each production zero, one or more semantic actions as needed;
3. simulate the parser actions for the given terminal string and execute the corresponding semantic actions on the symbol table. Verify that the resulting symbol table is exactly the previous one.

Exercise 2. Syntax-Directed Translation in top-down parsing

An Syntax-Directed Definition is *L-attributed* if each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production A -+ X_1X_2 - - X_n, and that there is an inherited attribute X_i.*a* computed by a rule associated with this production. Then the rule may use only:
    (a) Inherited attributes associated with the head A.
    (b) Either inherited or synthesized attributes associated with the occurrences of symbols X_1, X_2,. . . , X_{i-1} located to the left of X_i.
    (c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

In other words, the flow of attribute evaluation should be strictly from left to right.

Let consider the next L-attributed SDD defines a simple declaration D consisting of a basic type T followed by a list L of identifiers. T can be **int** or **float**. For each identifier on the list, the type is entered into the symbol-table entry for the identifier.

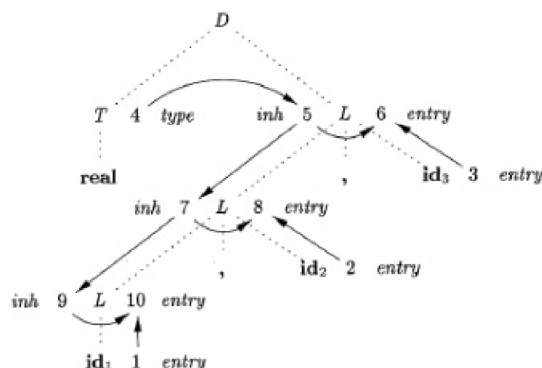
```
1) D -> T L              L.inh = T.type
2) T -> int              T.type = integer
3) T -> float            T.type = float
4) L -> L1 , id          L1.inh=L.inh
                         addType(id. entry, L.inh)
5) L -> id               add Type(id. entry, L.inh)
```

Non-terminal D represents a declaration, which, from production 1, consists of a type T followed by a list L of identifiers. T has one attribute, T.*type*, which is the type in the declaration D. Non-terminal L also has one attribute, which is called *inh* to emphasize that it is an inherited attribute. The purpose of L.*inh* is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol-table entries.

Production 4 passes L.*inh* down the parse tree. That is, the value L_1 .*inh* is computed at a parse-tree node by copying the value of L.*inh* from the parent of that node; the parent corresponds to the head of the production. Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments:

1. **id**.*entry*, a lexical value that points to a symbol-table object, and
2. L.*inh*, the type being assigned to every identifier on the list.

Here is an example of the dependency graph for the declaration **float** id_1 , id_2 , id_3 :



The attribute evaluation flow is left-to-right.

Use the previous example to build the annotated parse trees for the following expressions:

a) `int  a, b, c.`

b) `float w, x, y, z`

Exercise 3. Syntax-Directed Translation in both top-down and bottom-up parsing

A *LR-attributed grammar* is a special type of attribute grammars. It allows the attributes to be evaluated on LR parsing. As a result, attribute evaluation in LR-attributed grammars can be incorporated conveniently in bottom-up parsing. LR-attributed grammars are a subset of the L-attributed grammars, where the attributes can be evaluated in one left-to-right traversal of the abstract syntax tree. They are also a superset of the S-attributed grammars, which allow only synthesized attributes.

In yacc, a common hack is to use global variables to simulate some kind of inherited attributes (they are opposed to the bottom-up parsing mechanism) and thus LR-attribution.

zyacc (http://www.cs.binghamton.edu/~zdu/zyacc/doc/zyacc_4.html) is based on LR-attributed grammars.

A Pascal program can declare two integer variables a and b with the syntax

```
var a, b: int
```

This declaration might be described with the following grammar:

<VarDecl> -> **var** <IDList> **:** <TypeID>
<IDList> -> <IDList>**, ID**
| ***ID***
<TypeID> -> **int**
| **real**

where <IDList> derives a comma-separated list of variable names and <TypeID> derives a valid Pascal type. We have a typical case of attribute inheritance (the attribute value of <TypeID> should be inherited by the ID components of the <IDList> construct that appears before in the program text).

a. Write an attribute grammar that assigns the correct data type to each declared variable.

b. Write an ad hoc syntax-directed translation scheme that assigns the correct data type to each declared variable.

c. Can either scheme operate in a single pass over the syntax tree?

Hint. To evaluate and register the type of each ID in the symbol table during a bottom up parsing you may find it necessary to rewrite the grammar.