Evaluation of Semantic Actions in Predictive Non-Recursive Parsing

José L. Fuertes, Aurora Pérez
Dept. LSIIS
School of Computing. Technical University of Madrid
Madrid, Spain

Abstract— **To implement a syntax-directed translator, compiler designers always have the option of building a compiler that first performs a syntax analysis and then transverses the parse tree to execute the semantic actions in order. Yet it is much more efficient to perform both processes simultaneously. This avoids having to first explicitly build and afterwards transverse the parse tree, which is a time- and resource-consuming and complex process. This paper introduces an algorithm for executing semantic actions (for semantic analysis and intermediate code generation) during predictive non-recursive LL(1) parsing. The proposed method is a simple, efficient and effective method for executing this type of parser and the corresponding semantic actions jointly with the aid of no more than an auxiliary stack.**

I. INTRODUCTION

A parser uses a context-free grammar (*G*) to check if the input string syntax is correct. Its goal is to build the syntax tree for the analyzed input string. To do this, it applies the grammar rules. The set of valid strings generated by this grammar is the language (*L(G)*) recognized by the parser.

An LL(1) parser is built from an LL(1) grammar. The symbols of the grammar are input into a stack. The non-terminal symbol on the top of the stack and the current symbol of the input string determine the next grammar rule to be applied at any time.

A syntax-directed translator is built by defining attributes for the grammar symbols and semantic actions to compute the value of each attribute depending on others. This translator performs syntax analysis, semantic analysis, and code (intermediate or object) generation tasks.

Semantic action execution [1] can be easily integrated into several different parser types. But if you have designed a compiler with a predictive non-recursive LL(1) parser, you will find that attributes for grammar symbols that have been removed from the LL(1) parser stack are required to execute most of the semantic actions [2].

One possible solution is to build the parser tree and then transverse this tree at the same time as the semantic actions are performed. The attribute values are annotated in the tree nodes. Evidently, there is a clear efficiency problem with this solution. It also consumes an unnecessarily large quantity of resources (memory to store the whole tree, plus the node attributes, execution time…), not to mention the extra work on implementation. For this reason, a good approach is to evaluate the semantic actions at the same time as syntax analysis is performed [3] [4].

Semantic actions can be evaluated during LL parsing by extending the parser stack. The extended parser stack holds action-records for execution and data items (synthesize-records) containing the synthesized attributes for non-terminals. The inherited attributes of a non-terminal *A* are placed in the stack record that represents that non-terminal. On the other hand, the synthesized attributes for a non-terminal *A* are placed in a separate synthesize-record right underneath the record for *A* in the stack [5].

In this article, we introduce an algorithm for a top-down translator that provides a simpler, more efficient and effective method for executing an LL(1) parser and the corresponding semantic actions jointly with the aid of no more than an auxiliary stack.

The remainder of the article is organized as follows. Section II reviews the notions of top-down translators. Section III describes how the proposed top-down translator works, and section IV introduces an algorithm to implement this translator. Section V shows an example of how this method works. Finally, section VI outlines some conclusions.

II. RELATED WORK

This section reviews the concepts of top-down parsers and translation schemes that can be used to cover semantic and code generation aspects.

A. *Top-Down Parser*

A parser applies context-free grammar rules [6] to try to find the syntax tree of the input string. A top-down parser builds this tree from the root to the leaves. At the end of the analysis, the tree root contains the grammar's start symbol and the leaves enclose the analyzed string, provided this is correct.

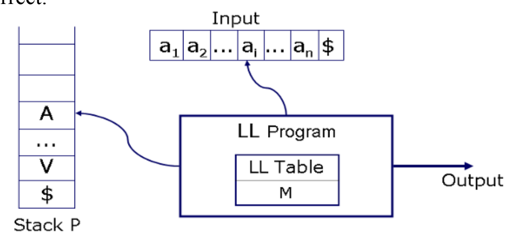

Fig. 1. Overview of an LL parser

T. Sobh (ed.), *Innovations and Advances in Computer Sciences and Engineering*,
DOI 10.1007/978-90-481-3658-2_86, © Springer Science+Business Media B.V. 2010

M	...	a_i	...	$
...	...	...	...	...
A	...	A→XYZ	...	...
...	...	...	...	...

Fig. 2. LL(1) parsing table.

Additionally, a compiler parser always has to produce the same parser tree for each input string. In the case of an LL(k) parser, the mechanism used to assure that there is only one rule applicable at any time is an LL(k) grammar. This grammar finds out which rule has to be applied by looking ahead at most *k* symbols in the input string. The simplest grammar of this type is LL(1). LL(1) finds out which rule to apply by looking no further than the first symbol in the input string.

An LL parser (Fig. 1) uses a stack (*P*) of grammar symbols and a table (*M*). The table (*M*) stores information on which rule to use to expand a non-terminal symbol on the top of the stack (*A*) depending on the current input symbol (a_i).

As initially configured the stack contains a symbol to indicate the bottom of stack (*$*) with the grammar's start symbol on top.

The rows of the LL(1) parsing table (Fig. 2) contain the non-terminal symbols. The table columns include the terminal symbols (set of valid input symbols) plus the end-of-string symbol (*$*). The table cells can contain a grammar production or be empty. If the parser accesses an empty cell, there is a syntax error in the input string. By definition, an LL can evidently never have more than one rule per cell.

If there is a non-terminal symbol, *A*, on top of the stack, the parser inspects the current input symbol, a_i, and looks up the matching table cell, $M [A, a_i]$, as shown in Fig. 1. This cell contains the rule to be applied (see Fig. 2). Then the non-terminal symbol *A* that was on top of stack *P* is removed and replaced by the right side of the applied rule. The symbol that was in the left-most position on the right side of the production (in this case *X*) is now on top of the stack (see Fig. 3). This is the next symbol to be expanded.

If there is a terminal symbol on top of the stack, it must match the current input symbol. If they are equal, the parser takes out the symbol on top of the stack and moves ahead to the next symbol in the input string. Otherwise, the parser discovers that the syntax of the input string is incorrect.

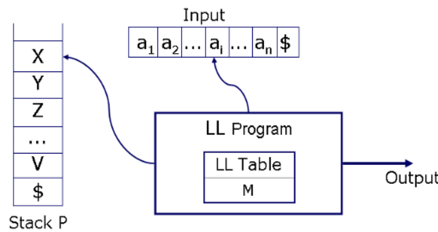

Fig. 3. Configuration of the parser after expanding rule *A* → *X Y Z*.

B. Translation Schemes

A context-free grammar accounts for the syntax of the language that it generates but cannot cover aspects of the semantics of this language. For example, let rule (1) be:

S → id := E (1)

Rule (1) reproduces a language's assignation sentence syntax perfectly. But it is no use for checking whether the expression and identifier types are compatible or, conversely, the programmer is trying to assign an invalid value to that identifier.

A translation scheme is a context-free grammar in which attributes are associated with the grammar symbols and semantic actions are inserted within the right sides of productions [1]. These semantic actions are enclosed between brackets *{ }*. The attributes in each production are computed from the values of the attributes of grammar symbols involved in that production [7].

So, a translation scheme can include semantic information by defining:

- as many attributes as semantic aspects need to be stated for each symbol
- semantic actions that compute attribute values.

For rule (1), for example, the type attribute would be used for both the identifier (*id*) and the expression (*E*), and it would need to check that *id.type* is equal to or compatible with *E.type*.

There are two kinds of attributes: synthesized and inherited [8]. An attribute is synthesized if its value in a tree node depends exclusively on the attribute values of the child nodes. In any other case, it is an inherited attribute. In rule (2), for example, *A.s* is synthesized and *Y.i* is inherited.

A → X {Y.i:= g (A.i, X.s)} Y Z {A.s:= f (X.s, Y.i)} (2)

An L-attributed translation scheme assures that an action never refers to an attribute that has not yet been computed. An L-attributed translation scheme uses a subset of attributes [9] formed by:

- all the synthesized attributes
- inherited attributes for which the value of an attribute in a node is computed as a function of the inherited attributes of the parent and/or attributes of the sibling nodes that are further to the left than the node.

Whereas the *Y.i* and *A.s* attributes in rule (2) above meet this requirement, the attribute *X.i* would not if the rule included the semantic action *X.i:= h (A.s, Z.i)*.

III. Proposed Top-Down Translator

In this section we introduce the design of the proposed top-down translator that can output the translation (the intermediate or object code in the case of a compiler) at the same time as it does predictive non-recursive parsing. This saves having to explicitly build the annotated parse tree and then transverse it to evaluate the semantic actions (perhaps also having to build the dependency graph [10] to establish the evaluation order).

We use an L-attributed translation scheme as a notation for specifying the design of the proposed translator. To simplify translator design, we consider the following criteria:

Criterion 1. A semantic action computing an inherited symbol attribute will be placed straight in front of that symbol.

Criterion 2. An action computing a synthesized attribute of a symbol will be placed at the end of the right side of the production for that symbol.

For example, (3) would be a valid rule:

$$X \rightarrow Y_1\ Y_2\ \{Y_3.i:=f(X.i,\ Y_1.s)\}\ Y_3\ Y_4\ Y_5\ \{X.s:=g(Y_3.i,\ Y_4.s)\} \quad (3)$$

To generate the proposed top-down translator the LL(1) parser is modified as follows. First, stack *P* is modified to contain not only grammar symbols but also semantic actions. Second, a new stack (*Aux*) is added. This stack will temporally store the symbols removed from stack *P*. Both stacks are extended to store the attribute values (semantic information).

Let us now look at how the attribute values will be positioned in each stack. To do this, suppose that we have a generic production $X \rightarrow \alpha$. This production contains semantic actions before and after each grammar symbol, where $\alpha \equiv \{1\}\ Y_1\ \{2\}\ Y_2 ...\ \{k\}\ Y_k\ \{k+1\}$.

Fig. 4 shows the parser stack *P* and the auxiliary stack *Aux*, both augmented to store the symbol attributes. For simplicity's sake, suppose that each grammar symbol has at most one attribute. If it had more, each position in the extended stacks would be a register with one field per attribute.

Suppose that these stacks are configured as shown in Fig. 4, with semantic action *{i}* at the top of stack *P*. This means, as we will see from the algorithm presented in section 4, that this semantic action should be executed. There is a pointer to the top of each stack. After executing the semantic action *{i}*, there will be another pointer to the new top (*ntop*) of stack *P*.

Because of the above-mentioned Criterion 1, the semantic action *{i}* uses an inherited attribute of *X* and/or any attribute of any symbol Y_j ($1 \leq j < i$) on the right side of the production to compute the inherited attribute of Y_i. If $i = k + 1$, the action *{i}* computes the synthesized attribute of *X*, because of Criterion 2. The following then applies.

- *Case 1*. The semantic action *{i}* computes the inherited attribute of Y_i.
  The symbol Y_i will be in stack *P*, right underneath action *{i}*, which is being executed. Thus, Y_i will be the new top (*ntop*) of stack *P* at the end of this execution. The reference to an inherited attribute of Y_i can be viewed as an access to stack *P* and, specifically, position *P [ntop]*.
- *Case 2*. The semantic action *{i}* contains a reference to an attribute of *X*.
  By definition of the L-attributed translation scheme, this will always be a reference to an inherited attribute of *X*. Only if $i = k + 1$ will there be a reference to a synthesized attribute of *X*. As *X* will have already been removed from stack *P* when the rule $X \rightarrow \alpha$ was applied, the symbol *X* will have been entered in stack *Aux*. All the grammar symbols Y_1, Y_2... Y_{i-1} (preceding the semantic action *{i}*) will be on top of *X*. These symbols will have been removed from *P* and inserted into *Aux*. Then any reference in *{i}* to an attribute of *X* can be viewed as an access to stack *Aux*, specifically, position *Aux [top – i + 1]*.
- *Case 3*. The semantic action *{i}* contains a reference to an attribute of some symbol of α.
  By definition of the L-attributed translation scheme, this attribute will necessarily belong to a symbol positioned to the left of action *{i}*, i.e. to one of the symbols Y_1, Y_2... Y_{i-1}. These symbols will have already been moved from stack *P* to stack *Aux*. Then any reference to an attribute of any of these symbols of α can be viewed as an access to stack *Aux* taking into account that Y_{i-1} will be on *Aux [top]*, Y_{i-2} will be on *Aux [top – 1]*... Y_1 will be on *Aux [top – i + 2]*.

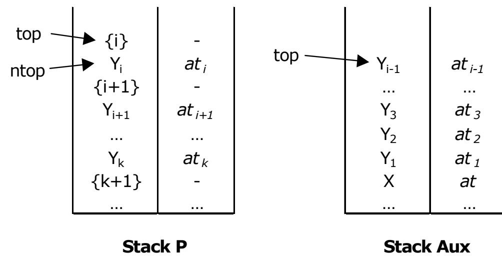

Fig. 4. Translator stacks *P* and *Aux* after applying $X \rightarrow \alpha$ while processing the elements of α.

The translator is implemented by programming the semantic actions and inserting them into the parser code. These semantic actions were written in terms of grammar symbols and attributes in the translation scheme. They now have to be rewritten in terms of accesses to the exact stack positions containing the values of the symbol attributes referenced in each semantic action.

The two criteria are designed merely to simplify the translator design. But the method would also work provided that the semantic action computing an inherited attribute of a symbol is located before, but not necessarily straight in front of, that symbol (Criterion 1). It would also be operational if the semantic action computing a synthesized attribute of, the symbol located on the left side of the production (*X.s*) is not at the right end of the production (Criterion 2) but depends exclusively on symbol attributes to its left. Therefore, we could also use rule (4) instead of rule (3). In the first semantic action, attribute $Y_3.i$ will be positioned in the middle of stack *P*, specifically *P [ntop – 1]* in this case. The second semantic action is also a valid action because *X.s* does not depend on Y_5. The referenced attributes will be in stack *Aux* ($Y_3.i$ at *Aux [top – 1]*, $Y_4.s$ at *Aux [top]* and *X.s* at *Aux [top – 4]*.

$$X \rightarrow Y_1\ \{Y_3.i:= f\ (X.i,\ Y_1.s)\}\ Y_2\ Y_3\ Y_4\ \{X.s:= g\ (Y_3.i,\ Y_4.s)\}\ Y_5 \quad (4)$$

IV. TOP-DOWN TRANSLATOR ALGORITHM

Having established the principles of the proposed top-down translator, we are now ready to present the algorithm. This algorithm is an extended version of the table-driven predictive non-recursive parsing algorithm that appears in [1].

The algorithm uses a table *M'*. This table is obtained from the LL parser table *M* by substituting the rules of the grammar *G* for the translation scheme rules (which include the modified semantic actions for including stack accesses instead of attributes). Then the proposed top-down translator algorithm is described as follows:

Input. An input string ω, a parsing table M for grammar G and a translation scheme for this grammar.

Output. If ω is in $L(G)$, the result of executing the translation scheme (translation to intermediate or object code); otherwise, an error indication.

Method. The process for producing the translation is:

1. Each reference in a semantic action to an attribute is changed to a reference to a position in the stack (*P* or *Aux*) containing the value of this attribute. Then the translation scheme is extended by adding a new semantic action at the end of each production. This action pops as many elements from the stack *Aux* as grammar symbols there are in the right side of the production. Finally, this modified translation scheme is incorporated into table *M*, leading to table *M'*.
2. Initially, the configuration of the translator is:
   - *$S* is in stack *P*, with *S* (the start symbol of *G*) on top,
   - the stack *Aux* is empty, and
   - $\omega\$$ is in the input, with *ip* pointing to its first symbol.
3. **Repeat**
   Let *X* be the symbol on top of stack *P*
   Let *a* be the input symbol that *ip* points to
   If *X* is a terminal **Then**
     If *X = a* **Then**
       Pop *X* and its attributes out of stack *P*
       Push *X* and its attributes onto stack *Aux*
       Advance *ip*
     Else Syntax-Error ()
     If *X* is a non-terminal **Then**
       If $M[X, a] = X \rightarrow \{1\}\ Y_1\ \{2\}\ Y_2 \ldots \{k\}\ Y_k\ \{k+1\}$
         Then
         Pop *X* and its attributes out of stack *P*
         Push *X* and its attributes onto stack *Aux*
         Push $\{k+1\}, Y_k, \{k\} \ldots Y_2, \{2\}, Y_1, \{1\}$ onto stack *P*, with *{1}* on top
       Else Syntax-Error ()
     If *X* is a semantic action *{i}* **Then**
       Execute *{i}*
       Pop *{i}* out of stack *P*
   Until *X = $* and *Aux = S*

V. EXAMPLE

To illustrate how the method works let us use a fragment of a C/C++ grammar (Fig. 5) designed to declare local variables. Fig. 6 shows the translation scheme for this grammar.

Based on the translation scheme, apply step 1 of the algorithm described in section 4 to build the modified version of the translation scheme (see Fig. 7). In Fig. 7, references to the inherited attributes *L.type* in rule (*1*) and rule (*5*) and *R.type* in rule (*4*) have been changed to references to new top (*ntop*) of stack *P*. References to other attributes have been replaced with references to stack *Aux*. New semantic actions (calls to the *Pop* function) are included to remove symbols that are no longer needed from stack *Aux*. The number of symbols to be removed is the number of grammar symbols on the right side of the production. This number is passed to the *Pop* function.

Table 1 shows table *M'* for the modified translation scheme that includes references to stack positions instead of attributes. We have numbered the semantic actions with the production number and, if necessary, with a second digit showing the order of the semantic action inside the production. For instance, action *{1.1}* represents *{P [ntop]:= Aux [top]}*, the first action of production *(1)*, whereas action *{1.2}* represents *{Pop (3)}*, the second action of production *(1)*.

To illustrate how the translator works, consider the input: '*float x, y;*'. This string is tokenized by the scanner as the input string $\omega \equiv$ *float id, id;*.

```
(1)  D→ T L ;
(2)  T→ int
(3)  T→ float
(4)  L→ id R
(5)  R→ , L
(6)  R→ λ
```

Fig. 5. Grammar for C/C++ variables declaration.

```
(1)  D→ T {L.type:= T.type}
         L ;
(2)  T→ int {T.type:= integer}
(3)  T→ float {T.type:= float}
(4)  L→ id {insertTypeST (id.ptr, L.type);
            R.type:= L. type}
         R
(5)  R→ , {L.type:= R.type}
         L
(6)  R→ λ {}
```

Fig. 6. Translation scheme for C/C++ variables declaration.

```
(1)  D→ T {P[ntop]:= Aux[top]}
         L ; {Pop (3)}
(2)  T→ int {Aux[top-1]:= integer; Pop (1)}
(3)  T→ float {Aux[top-1]:= float; Pop (1)}
(4)  L→ id {insertTypeST (Aux[top], Aux[top-1]);
            P[ntop]:= Aux[top-1]}
         R {Pop (2)}
(5)  R→ , {P[ntop]:= Aux[top-1]}
         L {Pop (2)}
(6)  R→ λ {}
```

Fig. 7. Modified translation scheme for C/C++ variables declaration including references to stack positions.

TABLE 1
TABLE M' FOR THE MODIFIED TRANSLATION SCHEME ILLUSTRATED IN FIG. 7.

M'	id	int	float	;	,	$
D		D → T {1.1} L ; {1.2}	D → T {1.1} L ; {1.2}			
T		T → int {2}	T → float {3}			
L	L → id {4.1} R {4.2}					
R				R → λ	R → , {5.1} L {5.2}	

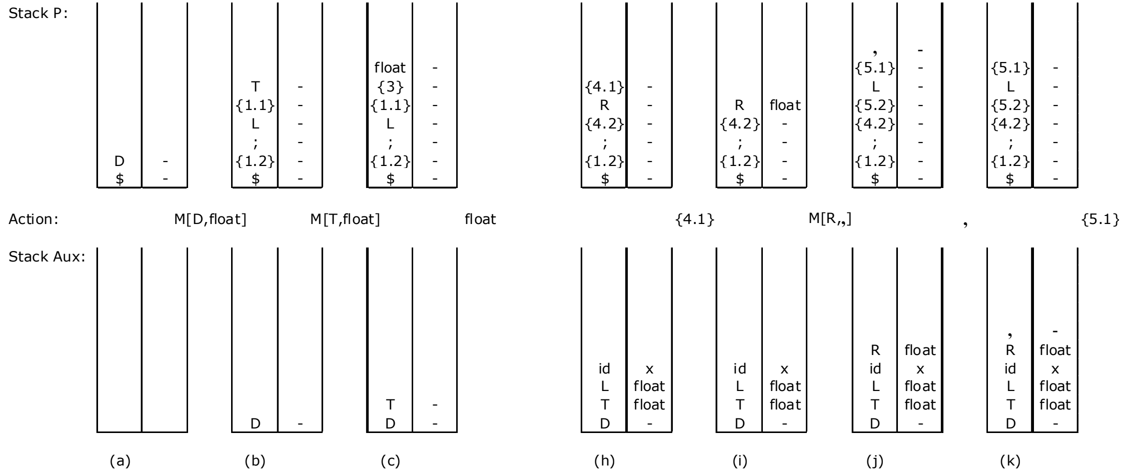

Fig. 8. First stack configurations (the input string is '*float x, y;*').

Fig. 10. Stack configurations 8 to 11 ('*, y;*' is in the input).

The stacks are initialized (Fig. 8(a)) with *$* and the grammar start symbol (*D*). Figs. 8 to 13 illustrate the different configurations of the extended stacks *P* and *Aux* (stack *P* is positioned above stack *Aux* throughout, and the action taken is stated between the two stacks).

As *D* (a non-terminal) is on top of stack *P* and *float* is the first symbol of ω, check *M [D, float]*. This gives the production $D \rightarrow T\,\{1.1\}\,L\,;\,\{1.2\}$. Therefore, move *D* from stack *P* to stack *Aux* and push the right side of the production onto stack *P* (Fig. 8(b)).

As shown in Fig. 8(c) we find that there is a terminal *float* on top of stack *P*. As this matches the current input symbol, transfer it from stack *P* to stack *Aux* and move *ip* ahead to point to the next input symbol (*id*).

The next element on top of stack *P* is the semantic action *{3}* to be executed. This action is *{Aux [top – 1]:= float; Pop (1)}*. First, insert *float* as the attribute value of symbol *T* (the second symbol from the top of stack *Aux*). Then execute the *Pop* function, which removes one element (*float*) from the top of stack *Aux* (Fig. 9(e)).

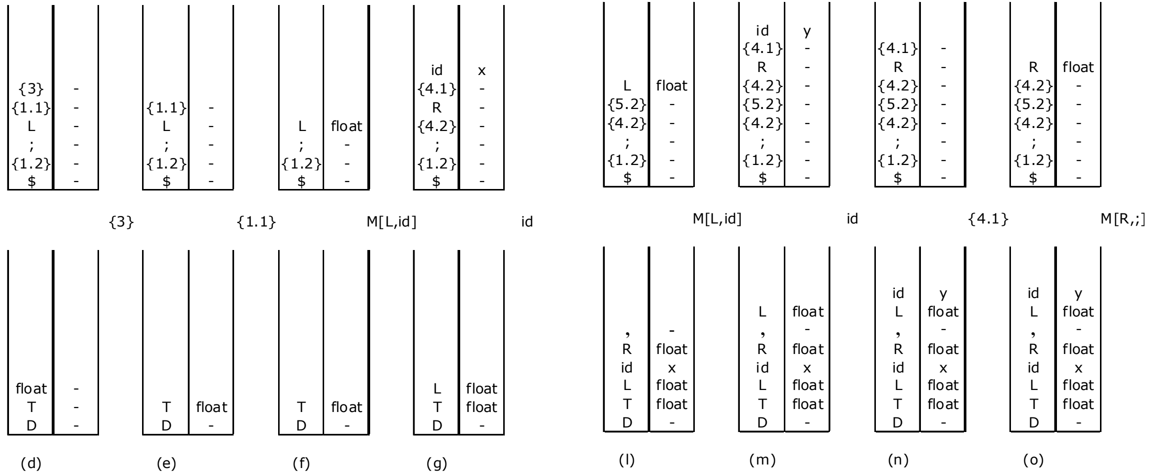

Fig. 9. Stack configurations 4 to 7 ('*x, y;*' is in the input).

Fig. 11. Stack configurations 12 to 15 ('*y ;*' is in the input).

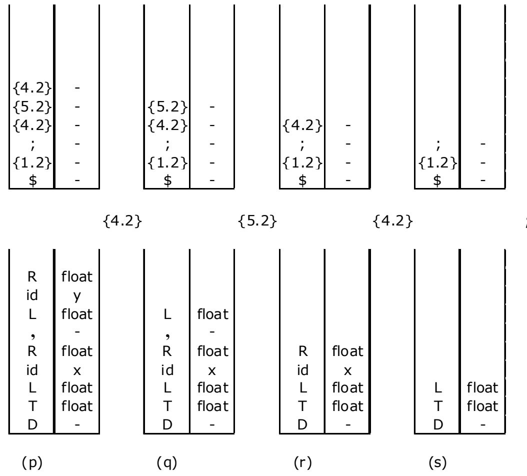

Fig. 12. Stack configurations 16 to 19 (';' is in the input).

As shown in Fig. 9(g), the *id* token has an attribute gathered directly from the scanner. This attribute is handled as a reference to the symbol table entry for this identifier. We represent the attribute using just the name of the identifier in the source code (*x*). Later the semantic action *{4.1}* is on top of stack *P* (Fig. 10(h)). Its execution copies *float* from stack *Aux* to stack *P* (*{P [ntop]:= Aux [top – 1]}*) as the attribute value of symbol *R*. The analysis continues as shown in Figs. 10 to 12.

In addition, two actions executed in this example (specifically, semantic action *{4.1}* executed in Fig. 10(h) and Fig. 11(n)) will have included type information about the *x* and *y* float identifiers in the compiler's symbol table.

Fig. 13(t) shows the execution of action *{1.2}* that removes three symbols from the stack *Aux*. Fig. 13(u) represents the algorithm exit condition.

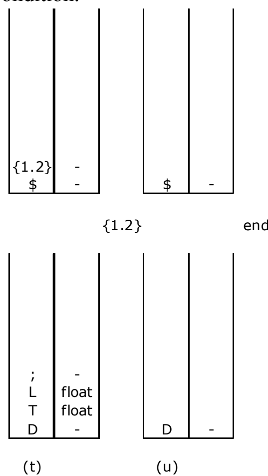

Fig. 13. Final stack configurations having read the whole input string.

VI. Conclusions

We have introduced a simple method and an algorithm to manage the evaluation of semantic actions in predictive non-recursive top-down LL(1) parsing. The main advantage of this method is that it can evaluate semantic actions at the same time as it parses. The main goal of this simultaneous execution is to save compiler resources (including both execution time and memory), since a compiler of this kind no longer needs to explicitly build a complete tree parser.

This method has been taught in a compilers course at the Technical University of Madrid's School of Computing for the last 6 years. As part of this course, students are expected to build a compiler for a subset of a programming language. About one third of students used this method, with very encouraging results. The method has proved to be easy to implement and understand.

The two criteria imposed in section 3 are merely to simplify the translator design. But the method is general and can be applied to any L-attributed translation scheme for an LL(1) grammar. Additionally, the tests run by students from our School on their compilers have shown that it is an efficient and simple way to perform the task of top-down syntax-directed translation.

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers. Principles, Techniques and Tools*, Addison-Wesley, 1985.

[2] R. Akker, B. Melichar, J. Tarhio, "Attribute Evaluation and Parsing", *Lecture Notes in Computer Science*, 545, Attribute Grammars, Applications and Systems, 1991, pp. 187-214.

[3] T. Noll, H. Vogler, "Top-down Parsing with Simultaneous Evaluation of Noncircular Attribute Grammars", *Fundamenta Informaticae*, 20(4), 1994, pp. 285-332.

[4] K. Müller, "Attribute-Directed Top-Down Parsing", *Lecture Notes in Computer Science*, 641, Proc. 4th International Conference on Compiler Construction, 1992, pp. 37-43.

[5] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, *Compilers. Principles, Techniques and Tools*, 2nd ed., Addison-Wesley, 2007.

[6] N. Chomsky, "Three models for the description of language", *IRE Transactions on Information Theory*, 2, 1956, pp. 113-124.

[7] T. Tokuda, Y. Watanabe, *An attribute evaluation of context-free languages*, Technical Report TR93-0036, Tokyo Institute of Technology, Graduate School of Information Science and Engineering, 1993.

[8] D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Lagendoen, *Modern Compiler Design*, John Wiley & Sons, 2000.

[9] O.G. Kakde, *Algorithms for Compiler Design*, Laxmi Publications, 2002.

[10] S.S. Muchnick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997.