COMPILER DESIGN

Syntax-Directed Translation

Syntax-directed translation: analysis and synthesis

- Translation process driven by the syntactic structure of the program, as generated by the parser.
- In syntax-directed translation, the semantic analysis and translation steps of the compilation process is divided in two parts:
  - analysis (syntactic, semantic)
  - synthesis (translation and optimization)
- The semantic analysis becomes the link between analysis and synthesis: translation (synthesis) is conditional to positive semantic analysis.
- The syntax-directed translation process is inducing very strong coupling between the syntax analysis phase, the semantic checking phase, and the translation phase.

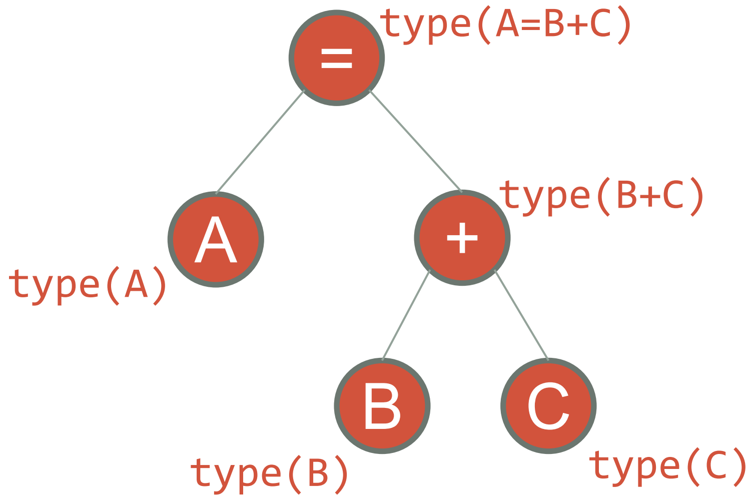
Syntax-directed translation: semantic actions, semantic values, attribute migration

- ***Semantic actions*** are integrated in the parsing process.
- Semantic actions are functions whose goal is to accumulate and/or process semantic information about the program as the parsing is done.
- Different pieces of information are gathered during the parsing of different syntactical rules.
- These pieces of information, named ***semantic values***, or ***semantic attributes*** are gathered and accumulated in ***semantic records*** until they can be processed by further semantic actions.
- Some of the information that must be gathered spans over the application of several syntactical rules.
- This raises the need to migrate this information across the parse tree, a process known as ***attribute migration***.

Syntax-directed translation: semantic actions

- Some semantic actions (implemented as ***semantic routines***) do the analysis phase by performing semantic aggregation and/or checking in productions that need such aggregation and/or checks, depending on the semantic rules defined by the language specification, e.g. *type checking/aggregation.*

  - *`A = B + C;`*



- Semantic actions assemble information in order to validate and eventually generate a <u>meaning</u> (i.e. translation) for the program elements generated by the productions.
- They are the starting point for translation (synthesis).
- Thus, the semantic routines are the <u>heart</u> of the compiler.

Syntax-directed translation: relationship to parse trees, nodes, and parsing methods

- Conceptually, semantic actions are associated with parse tree nodes:
  - Semantic actions at the ***leaves*** of the tree typically *gather* semantic information, either from the token the node represents, often with the help from a lookup in the *symbol table* (i.e. to get the type of an identifier).
  - Semantic actions in ***intermediate nodes*** typically *aggregate*, *validate*, or *use* semantic information and pass the information up in the tree (i.e. perform *attribute migration*).

- As parse trees are also related to grammar rules, so are semantic actions. Semantic actions are inserted in the right-hand sides of the grammar:
  - In recursive-descent predictive parsing, they are represented by ***function calls*** inserted in the parsing functions, attribute migration is performed by using ***parameters*** passed and returned between parsing functions.
  - In table-driven parsers, semantic action symbols are inserted in the right-hand-sides and ***pushed onto the stack***. Popping a semantic action symbol from the stack triggers the corresponding semantic action. Attribute migration is made by way of the parsing stack. Some parsers may have to rely on a separate ***semantic stack***.
  - Alternatively, recursive-descent predictive parsers can also use a semantic stack instead of local variables and parameter passing/return values.

Syntax-directed translation and attribute grammars

- Semantic routines can be formalized using attribute grammars.
- Attribute grammars augment ordinary context-free grammars with attributes that represent semantic properties such as type, value or correctness used in semantic analysis (checking) and code generation (translation).

- It is useful to keep checking and translation facilities distinct in the semantic routines' implementation.
- Semantic checking is machine-independent and code generation is not, so separating them gives more flexibility to the compiler (front/back end).

Attributes

- An ***attribute*** is a property of a programming language construct, including *data type, value, memory location/size*, *translated code,* etc.
- Implementation-wise, they are also called ***semantic records***.

- The process of computing and assigning the value of an attribute is called ***binding***. *Static binding* concerns binding that can be done at compile-time, and *dynamic binding* happens at run-time, e.g. for polymorphism.
- In our project, we are concerned solely on static compile-time binding.

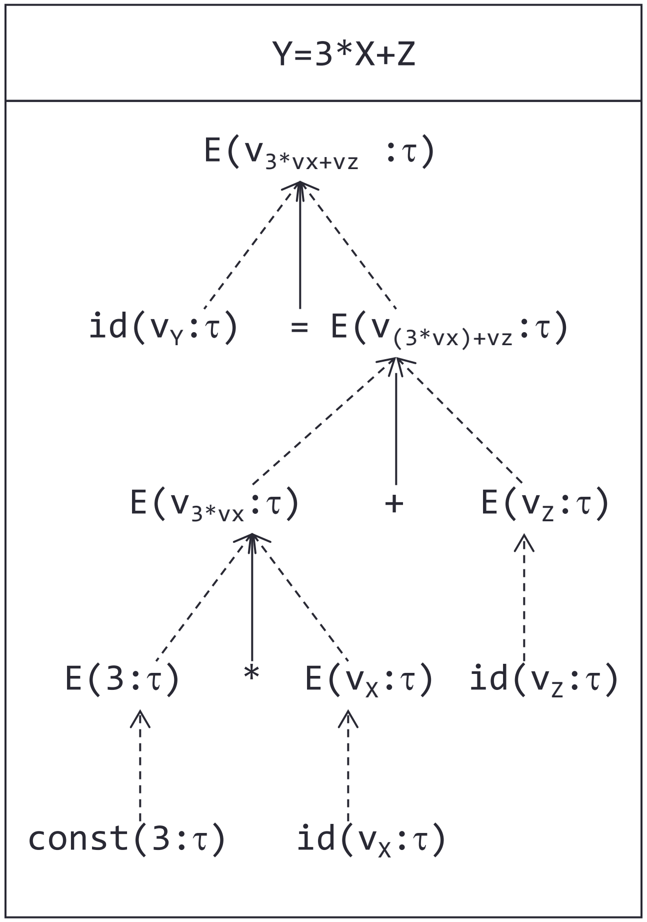
Attributes migration

- Static attribute binding is done by *gathering*, *propagating*, and *aggregating* attributes while traversing the parse tree.
- Attributes are *gathered* at tree leaves, *propagated* across tree nodes, and *aggregated* at some parent nodes when additional information is available.
- This can be done as the program is being parsed using *syntax-directed translation*.

- <u>Synthetized attributes</u> : attributes gathered from a child in the syntax tree
- <u>Inherited attributes</u> : attributes gathered from a sibling in the syntax tree

Example: Semantic rules and attribute migration

$E_1 \rightarrow id = E_2$	$E_= : \dfrac{[E_2: v:\tau]\ [(id:\tau) \in defVar]}{[id.val \leftarrow v]\ [E_1.val: \tau \leftarrow v]}$
$E_1 \rightarrow E_2 * E_3$	$E_* : \dfrac{[E_2: v_2:\tau]\ [E_3: v_3:\tau]}{[E_1.val: \tau \leftarrow v_2 * v_3]}$
$E_1 \rightarrow E_2 + E_3$	$E_+ : \dfrac{[E_2: v_2:\tau]\ [E_3 : v_3:\tau]}{[E_1.val: \tau \leftarrow v_2 + v_3]}$
$E \rightarrow id$	$E_{id} : \dfrac{[(id:\tau) \in defVar]}{[(E.val: \tau) \leftarrow id.val]}$
$E \rightarrow const$	$E_{const} : \dfrac{[const : v:\tau]}{[(E.val: \tau) \leftarrow v]}$

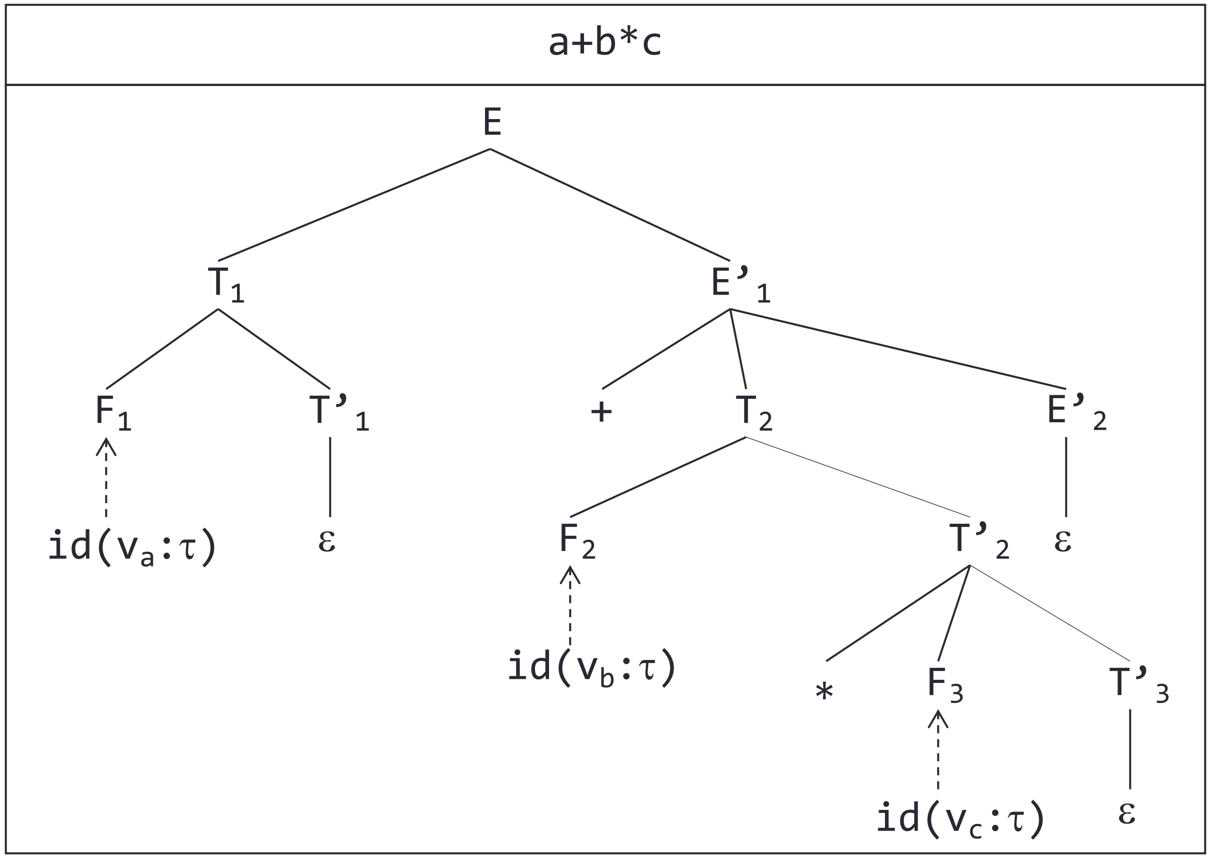


- Here, semantic information only flows upwards in the tree.
- All aggregated semantic information is available on the same subtree.

Example 2: attribute migration

- Problems arise when rules are factorized:

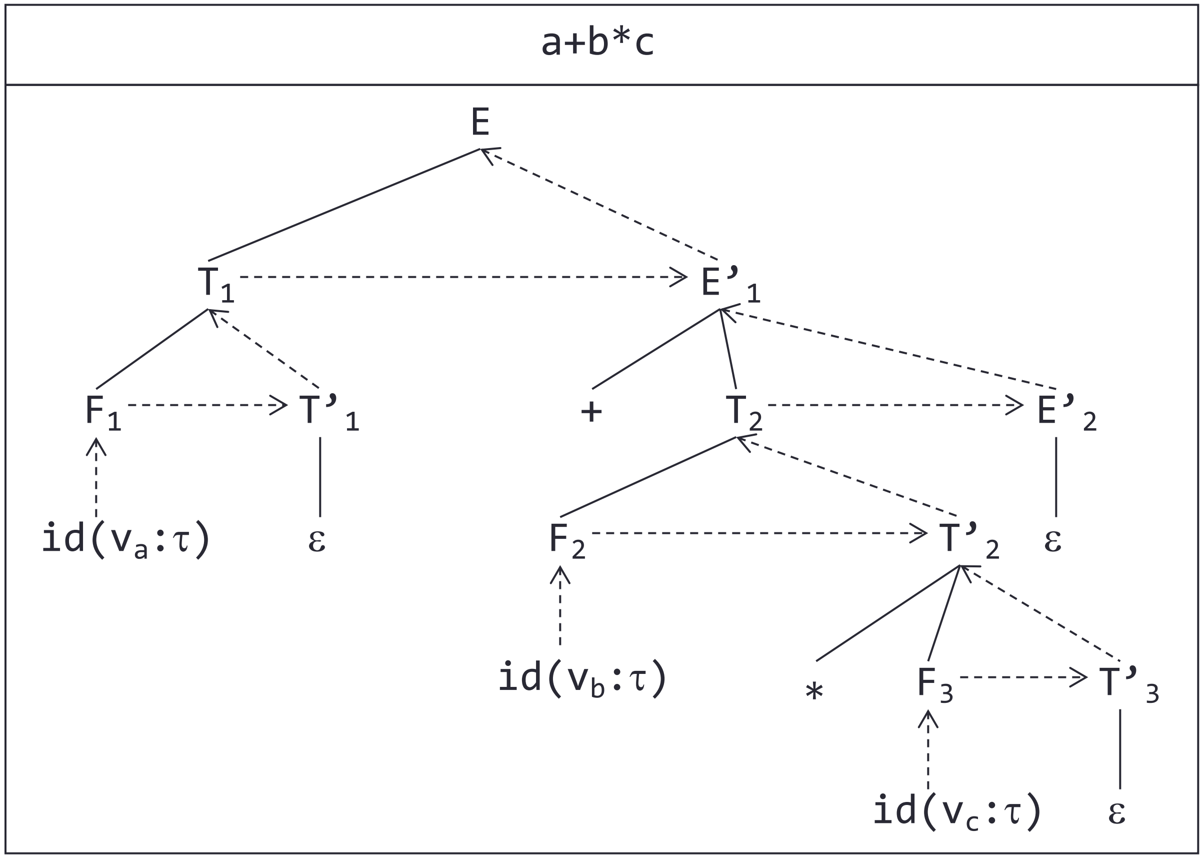
```
E  → TE’
E’ → +TE’|ε
T  → FT’
T’ → *FT’|ε
F  → id|const
```



Example 2: attribute migration

- Solution: migrate attributes sideways, i.e. ***inherited attributes***

```
E  → TE'
E' → +TE'|ε
T  → FT'
T' → *FT'|ε
F  → id|const
```



- Here, left operands are available on a different subtree, so they have to be migrated sideways across different subtrees in order to be aggregated.

Attribute migration: implementation in recursive-descent predictive parser

```
Parse(){
  semrec Es                         //semantic record created
                                    //before the call
  lookahead = NextToken()
  if (E(Es);Match('$'))             //passed as a reference
                                    //to parsing functions
                                    //that will compute its value
    return(true);
  else
    return(false);
}
```

- `semrec` potentially represents any kind of semantic record.
- In this example, it represents semantic records that carry type information across the parsing of an expression.
- It could also be tree nodes that are created, migrated and grafted/adopted in order to construct an abstract syntax tree.
- It could also in fact be both, as the tree is created, information, such as type or symbol tables can be gathered as the parse is done or the tree is traversed.

Attribute migration: implementation in recursive-descent predictive parser

```
E(semrec &Es){
  semrec Ts,E's
  if (lookahead is in [0,1,(])
    if (T(Ts);E'(Ts,E's);)        // E' inherits Ts from T
      write(E->TE')
      Es = E's                    // Synthetised attribute sent up
      return(true)
    else
      return(false)
  else
    return(false)
}
```

- Each parsing function potentially defines its own semantic records used locally the processing of its own subtree.
- **`Ts,E's`** are semantic records produced/used by the **`T()`** and **`E'()`** functions and returned by them to the **`E()`** function.
- In the pseudo-code presented here, the semantic records are exposed to the calling functions by way of reference parameter passing.

Attribute migration: implementation in recursive-descent predictive parser

```
E'(semrec &Ti, type &E's){
  semrec Ts,E'2s
  if (lookahead is in [+])
    if (Match('+');T(Ts);E'(Ts,E'2s))    // E' inherits from T
      write(E'->TE')
      E's = semcheckop(Ti,E'2s)          // Semantic check & synthetized
                                         // attribute sent up
      return(true)
    else
      return(false)
  else if (lookahead is in [$,)]
    write(E'->epsilon)
    E's = Ti                             // Synth. attr. is inhertied
                                         // from T (sibling, not child)
                                         // and sent up
    return(true)
  else
    return(false)
}
```

- Some semantic actions will do some semantic checking and/or semantic aggregation, such as a tree node adopting a child node, or inferring the type of an expression from two child operands.

Attribute migration: implementation in recursive-descent predictive parser

```
T(semrec &Ts){
  semrec Fs, T's
  if (lookahead is in [0,1,(])
    if (F(Fs);T'(Fs,T's);)       // T' inherits Fs from F
      write(T->FT')
      Ts = T's                   // Synthetized attribute sent up
      return(true)
    else
      return(false)
  else
    return(false)
}
```

Attribute migration: implementation in recursive-descent predictive parser

```
T'(semrec &Fi, type &T's){
  semrec Fs, T'2s
  if (lookahead is in [*])
    if (Match('*');F(Fs);T'(Fs,T'2s))     // T' inherits from F
      write(T'->*FT')
      T's = semcheckop(Fi,T'2s)           // Semantic check and
                                          // synthetized attribute sent up
      return(true)
    else
      return(false)
  else if (lookahead is in [+,$,)]
    write(T'->epsilon)
    T's = Fi                              // Synthetized attribute is
                                          // inhertied from F sibling
                                          // and sent up the tree

    return(true)
  else
    return(false)
}
```

Attribute migration: implementation in recursive-descent predictive parser

```
F(semrec &Fs){
  semrec Es
  if (lookahead is in [id])
    if (Match('id'))
      write(F->id)
      Fs = gettype(id.name,table)         // Gets the attribute ``type''
                                          // from the symbol table and
                                          // sends it up the tree as Fs
      return(true)
    else
      return(false)
  else if (lookahead is in [(])
    if (Match('(');E(Es);Match(')'))
      write(F->(E))
      Fs = Es                             // Synthetized attribute from E
                                          // sent up the tree as attribute
                                          // of F
      return(true)
    else return(false)
  else return(false)
}
```

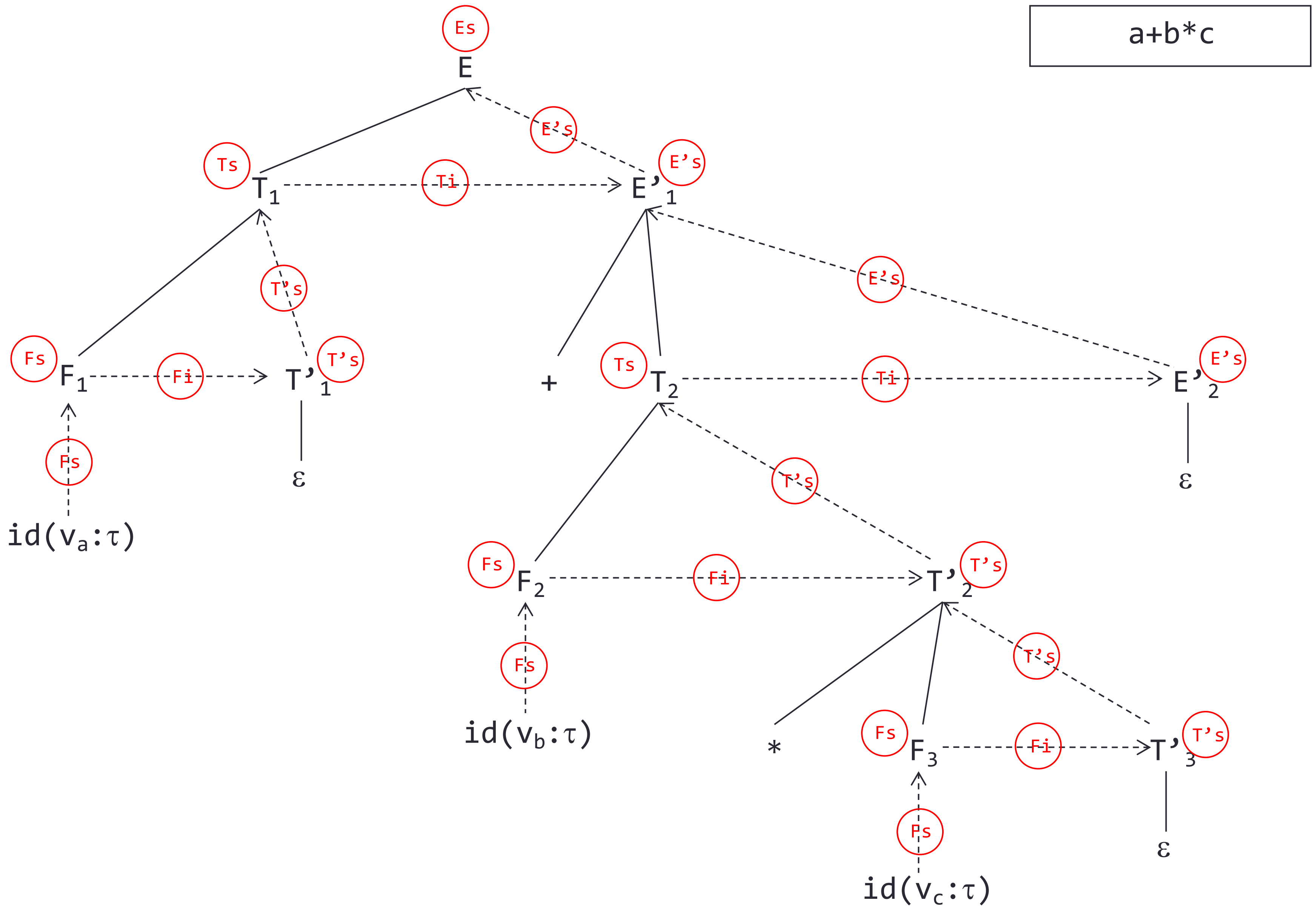
- Some semantic actions at the leaves will create new information to be propagated up.

Attribute migration: implementation in recursive-descent predictive parser

```
type semcheckop(type ti,type ts){
  if (ti == ts)
    return(ti)
  else
    return(typerror)
}
```

```
type gettype(name, table){
  if (name is in table)
    return (type)
  else
    return(typerror)
}
```

Attribute migration: example


a+b*c
Es
E
E's
Ts
T_1
Ti
E'_1
T's
Fs
F_1
Fi
T'_1
ε
id(v_a:τ)
+
T_2
E'_2
F_2
T'_2
id(v_b:τ)
*
F_3
T'_3
id(v_c:τ)