بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

قبل از ادامهٔ بحث و معرفی ***SDD***های ***L-attributed***، مروری اجمالی بر مفاهیم و مطالب جلسات قبل که حول موضوع ***SDD***ها و ***attribute grammar***ها بودند خواهیم داشت.

توضیح مختصری دربارهٔ *front end*

In the typical compiler, analysis and intermediate code generation mark the end of front end computation. The exact division of labor between the front end and the back end, however, may vary from compiler to compiler: it can be hard to say exactly where analysis (figuring out what the program means) ends and synthesis (expressing that meaning in some new form) begins (and there may be a "middle end" in between). Many compilers also carry a program through more than one intermediate form.

دربارهٔ ادغام پروسهٔ تجزیه با تحلیل معنایی و ساخت کد میانی

Compilers also vary in the extent to which semantic analysis and intermediate code generation are interleaved with parsing. ***With fully separated phases, the parser passes a full parse tree on to the semantic analyzer, which converts it to a syntax tree, fills in the symbol table, performs semantic checks, and passes it on to the code generator. With fully interleaved phases, there may be no need to build either the parse tree or the syntax tree in its entirety: the parser can call semantic check and code generation routines on the fly as it parses each expression, statement, or subroutine of the source.***

دربارهٔ ادغام پروسهٔ تجزیه با تحلیل معنایی و ساخت کد میانی

It is unclear whether interleaving semantic analysis with parsing makes a compiler simpler or more complex; it's mainly a matter of taste. If intermediate code generation is interleaved with parsing, one need not build a syntax tree at all (unless of course the syntax tree is the intermediate code). Moreover, it is often possible to write the intermediate code to an output file on the fly, rather than accumulating it in the attributes of the root of the parse tree. The resulting space savings were important for previous generations of computers, which had very small main memories. On the other hand, semantic analysis is easier to perform during a separate traversal of a syntax tree, because that tree reflects the program's semantic structure better than the parse tree does, especially with a top-down parser, and because one has the option of traversing the tree in an order other than that chosen by the parser.

☞ *It is conventional to say that the syntax of a language is precisely that portion of the language definition that can be described conveniently by a context-free grammar, while the semantics is that portion of the definition that cannot.*

☞ *In a simple SDD, every production has a single rule. In more complicated SDDs, each productions can have several rules.*

☞ *Tokens (terminals) often have intrinsic properties (e.g., the character-string representation of an identifier or the value of a numeric constant); in a compiler these are synthesized attributes initialized by the scanner (lexer).*

☞ *An attribute grammar is non-circular if it never leads to a parse tree in which there are cycles in the attribute flow graph—that is, if no attribute, in any parse tree, ever depends (transitively) on itself. As a general rule, practical attribute grammars tend to be non-circular. We can construct a topological sort of the attribute flow graph and then invoke rules in an order consistent with the sort.*

Example: *Suppose we wanted to count the elements of a comma-separated list:*

$L \longrightarrow$ `id` LT	✂ L.c := 1 + LT.c
$LT \longrightarrow$ `,` L	✂ LT.c := L.c
$LT \longrightarrow \epsilon$	✂ LT.c := 0

Here the rule on the first production sets the c (count) attribute of the left-hand side to one more than the count of the tail of the right-hand side. ***The rules are not allowed to refer to any variables or attributes outside the current production.***

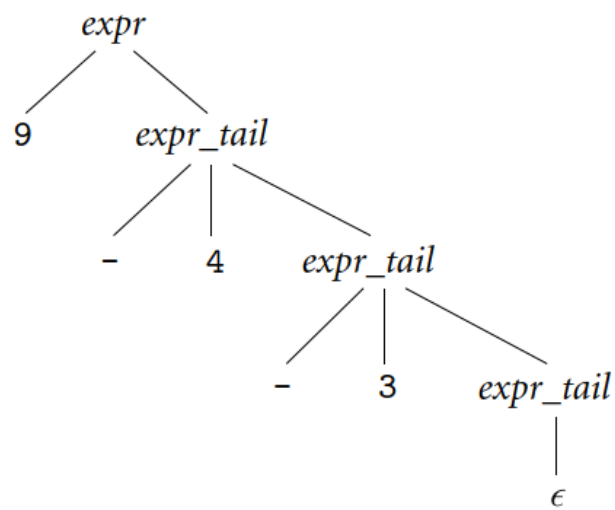
مثالی دیگر از بکارگیری خصیصه‌های موروثی

As a simple example of inherited attributes, consider the following fragment of an LL(1) expression grammar (here covering only subtraction):

expr ⟶ `const` *expr_tail*

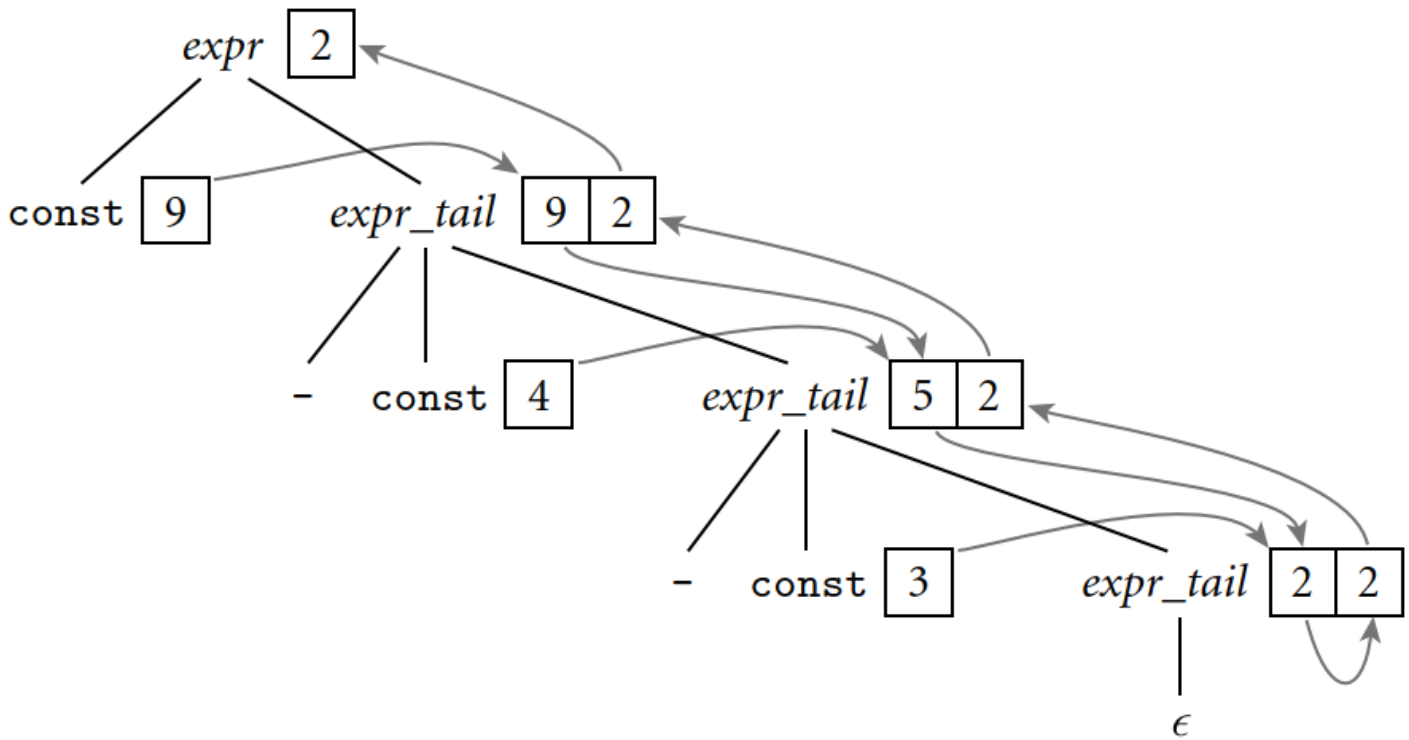
expr_tail ⟶ `-` `const` *expr_tail* | ϵ

For the expression `9 - 4 - 3`, we obtain the following parse tree:



If we want to create an attribute grammar that accumulates the value of the overall expression into the root of the tree, we have a problem: because subtraction is left associative, we cannot summarize the right subtree of the root with a single numeric value.

If, however, we are allowed to pass attribute values not only bottom-up but also left-to-right in the tree, then we can pass the 9 into the top-most *expr_tail* node, where it can be combined (in proper left-associative fashion) with the 4. The resulting 5 can then be passed into the middle *expr_tail* node, combined with the 3 to make 2, and then passed upward to the root:



To effect this style of decoration, we need the following attribute rules:

expr ⟶ `const` *expr_tail*
- ✂ expr_tail.st := const.val
- ✂ expr.val := expr_tail.val

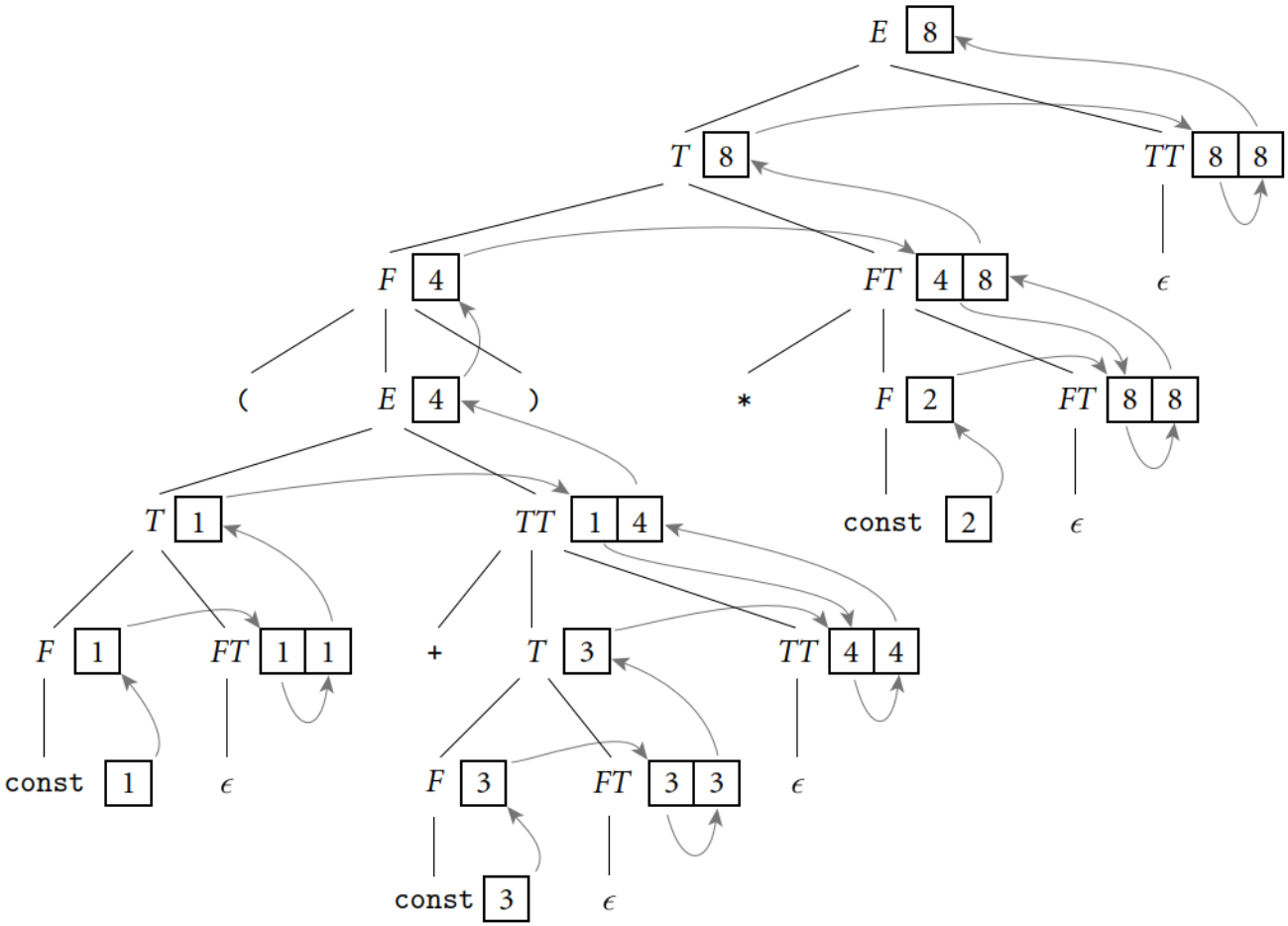
$expr\_tail_1$ ⟶ `-` `const` $expr\_tail_2$
- ✂ $expr\_tail_2$.st := $expr\_tail_1$.st − const.val
- ✂ $expr\_tail_1$.val := $expr\_tail_2$.val

expr_tail ⟶ ϵ
- ✂ expr_tail.val := expr_tail.st

In each of the first two productions, the first rule serves to copy the left context (value of the expression so far) into a "subtotal" (st) attribute; the second rule copies the final value from the right-most leaf back up to the root. In the *expr_tail* nodes of the picture, the left box holds the st attribute; the right holds val.

1. $E \longrightarrow T\ TT$
   ✂ TT.st := T.val ✂ E.val := TT.val
2. $TT_1 \longrightarrow$ `+` $T\ TT_2$
   ✂ TT_2.st := TT_1.st + T.val ✂ TT_1.val := TT_2.val
3. $TT_1 \longrightarrow$ `-` $T\ TT_2$
   ✂ TT_2.st := TT_1.st − T.val ✂ TT_1.val := TT_2.val
4. $TT \longrightarrow \epsilon$
   ✂ TT.val := TT.st
5. $T \longrightarrow F\ FT$
   ✂ FT.st := F.val ✂ T.val := FT.val
6. $FT_1 \longrightarrow$ `*` $F\ FT_2$
   ✂ FT_2.st := FT_1.st × F.val ✂ FT_1.val := FT_2.val
7. $FT_1 \longrightarrow$ `/` $F\ FT_2$
   ✂ FT_2.st := FT_1.st ÷ F.val ✂ FT_1.val := FT_2.val
8. $FT \longrightarrow \epsilon$
   ✂ FT.val := FT.st
9. $F_1 \longrightarrow$ `-` F_2
   ✂ F_1.val := − F_2.val
10. $F \longrightarrow$ `(` E `)`
    ✂ F.val := E.val
11. $F \longrightarrow$ `const`
    ✂ F.val := const.val

An attribute grammar for constant expressions based on an LL(1) CFG. In this grammar several productions have two semantic rules.



Decoration of a top-down parse tree for `(1 + 3) * 2`. Curving arrows again indicate attribute flow; the arrow(s) entering a given box represent the application of a single semantic rule. Flow in this case is no longer strictly bottom-up, but it is still left-to-right. At *FT* and *TT* nodes, the left box holds the `st` attribute; the right holds `val`.

Each semantic rule takes an arbitrary number of arguments (each of which must be an attribute of a symbol in the current production—no global variables are allowed), and each computes a single result, which must likewise be assigned into an attribute of a symbol in the current production. When more than one symbol of a production has the same name, subscripts are used to distinguish them. These subscripts are solely for the benefit of the semantic functions; they are not part of the context-free grammar itself.