Programming Languages — Attribute Grammar

Evaluating AST

Adding attributes to a grammar (and thus the AST), it is possible to describe much of the meaning of the AST. This permits automation of translation or interpretation.

BNF for simple expressions

The simplest view with [E]xpression, [T]erm, and [F]actor

1. This grammar is unambiguous and respects operator precedence among the four operators shown.
2. <symbol> is a non-terminal, :int: is an unsigned integer (a terminal symbol for our purposes).
3. -> marks a production rule, and | is the alternative rule indicator.
4. All other "punctuation" characters are character literals in the underlying language

```
<E> -> <E> + <T>
     | <E> - <T>
     | <T>

<T> -> <T> * <F>
     | <T> / <F>
     | <F>

<F> -> - <F>
     | ( <E> )
     | :uint:
```

Slight change: disambiguate multiple references to symbols of the same type within any rule

1. So, first rule, first clause has two <E> non-terminals. Label them <E1> and <E2> from left to right in the rule.
2. Continue with any rule that has two or more copies of a given symbol.

```
<E1> -> <E2> + <T>
      | <E2> - <T>
<E>  -> <T>

<T1> -> <T2> * <F>
      | <T2> / <F>
<T>  -> <F>

<F1> -> - <F2>
<F>  -> ( <E> )
      | :uint:
```

Adding Attributes

An attribute is a "field" added to a symbol.

If `E` has a `val` field (the expression has a value), that value is indicated with the dot notation: `E.val`, `E1.val`, etc.
Not all symbols need support a given attribute It makes little sense to think about `'('.val` A production rule is augmented with a semantic rule for determining attribute values:

```
Production Rule        Semantic Rule
<E1> -> <E2> + <T>  :: E1.val := E2.val + T.val
```

Attributed Grammar

```
Production Rule        Semantic Rule
<E1> -> <E2> + <T>  :: E1.val := E2.val + T.val
      | <E2> - <T>  :: E1.val := E2.val - T.val
<E>  -> <T>         ::  E.val := T.val

<T1> -> <T2> * <F>  :: T1.val := T2.val * F.val
      | <T2> / <F>  :: T1.val := T2.val / F.val
<T>  -> <F>         ::  T.val := F.val

<F1> -> - <F2>      :: F1.val := -F2.val
<F>  -> ( <E> )     ::  F.val := E.val
      | :uint:      ::  F.val := :uint:.val
```

Synthesized Attributes

Simple Parse

```
(1 + 2) * 3
Lexically: ( :uint:1: + :uint:2: ) * :uint:3:
Parse tree
                      <E>

                      <T>

              <T>      *       <F>

              <F>            :uint:3:

           (  <E>  )

        <E>    +    <T>

        <T>         <F>

        <F>        :uint:2:

       :uint:1:
```

Attributed Parse

`:uint:3:.val` is 3, `:uint:2:.val` is 2, and so on.

```
(1 + 2) * 3
Lexically: (:uint:1: + :uint:2:) * :uint:3:

Parse tree [synthesize val]

                     <E>[9]

                     <T>[9]

             <T>[3]   *       <F>[3]

             <F>[3]         :uint:3:

         (    <E>[3] )

        <E>[1] +    <T>[2]

        <T>[1]      <F>[2]

        <F>[1]     :uint:2:

       :uint:1:
```

Question: Parse and evaluate `1 + 2 * 3`; `2 / 5 * 20`

Inherited Attributes

Synthesized attributes go *up* the tree. Inherited attributes come *down* the tree. Remember the top-down grammar for expressions and its attribute grammar:

```
<E>  -> <T> <E'>
<E'> -> + <T> <E'>
      | - <T> <E'>
      | ε
<T>  -> <F> <T'>
<T'> -> * <F> <T'>
      | / <F> <T'>
      | ε
<F> -> - <F>
     | ( <E> )
     | :uint:
```

```
<E>   -> <T> <E'>       :: E'.in := T.val; E.val := E'.val
<E'1> -> + <T> <E'2>    :: E'2.in := E'1.in + T.val; E'1.val = E'2.val
<E'1> -> - <T> <E'2>    :: E'2.in := E'1.in - T.val; E'1.val = E'2.val
<E'>  -> ε              :: E'.val := E'.in
<T>   -> <F> <T'>       :: T'.in := F.val; T.val := T'.val
<T'1> -> * <F> <T'2>    :: T'2.in := T'1.in * F.val; T'1.val = T'2.val
<T'1> -> / <F> <T'2>    :: T'2.in := T'1.in * F.val; T'1.val = T'2.val
<T'>  -> ε              :: T'.val := T'.in
<F1>  -> - <F2>         :: F1.val := - F2.val
<F>   -> ( <E> )        :: F.val := E.val
       | :uint:         :: F.val := :uint:.val
```

`E'.in` brings in the value from the left. Notice that on an empty production value is just in and on the ones with operators, the operator sets the in value passed down.