

5. Semantics

Semantic Analysis

- Static semantics
- Dynamic semantics
- Attribute grammars
- Abstract syntax trees
- Putting theory into practice:
 - A Java interpreter of simple expressions
 - A Java translator of simple expressions to Lisp

Note: Study Chapter 4 of the textbook upto and including Section 4.3. Sections 4.4 to 4.6 are not required.

Static and Dynamic Semantics

- Syntax concerns the *form* of a valid program, while *semantic* concerns its *meaning*
- **Static semantic** rules are enforced by a compiler at compile time
 - Implemented in semantic analysis phase of the compiler
 - Context-free grammars are not powerful enough to describe certain rules, such as checking variable declaration with variable use
 - Examples: *Type checking; Identifiers are used in appropriate context; Check subroutine call arguments; Check labels*
- **Dynamic semantic** rules are enforced by the compiler by generating code to perform the checks
 - Examples: *Array subscript values are within bounds; Arithmetic errors; Pointers are not dereferenced unless pointing to valid object; A variable is used but hasn't been initialized*
 - Some languages (Euclid, Eiffel) allow programmers to add *explicit* dynamic semantic checks in the form of *assertion* e.g.

```
assert denominator not= 0
```
 - When a check fails at run time, an **exception** is raised

Attribute Grammars

- An *attribute grammar* links syntax with semantics
 - Every grammar production has a *semantic rule* with actions (e.g. assignments) to modify values of *attributes* of (non)terminals
 - A (non)terminal may have a number of *attributes*
 - Attributes have values that hold semantic information about the (non)terminal
 - General form:

<i>Production</i>	<i>Semantic rule</i>
$\langle A \rangle \rightarrow \langle B \rangle \langle C \rangle$	A.a := ...; B.a := ...; C.a := ...

A.a denotes attribute a of nonterminal $\langle A \rangle$

- Semantic rules are used by a compiler to enforce static semantics and/or to produce an abstract syntax tree while parsing token stream
- Can also be used to build simple language interpreters

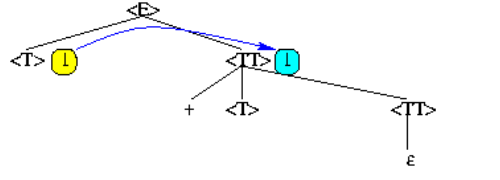
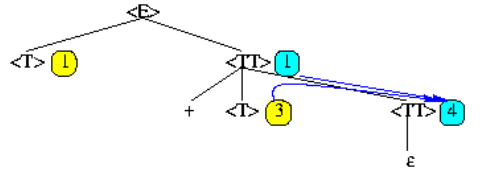
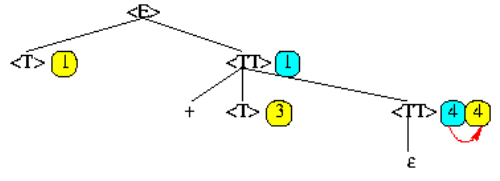
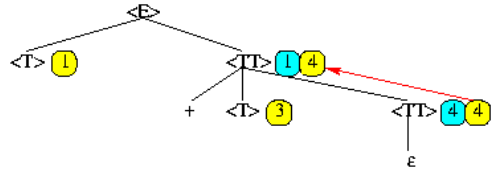
Example Attribute Grammar

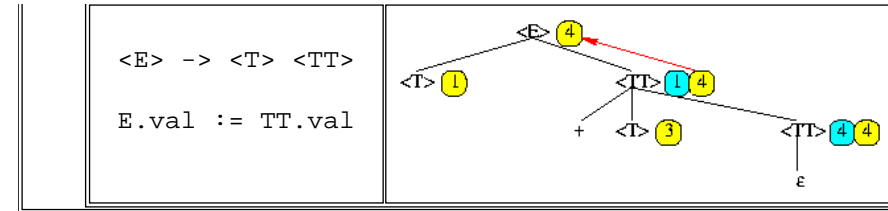
Example attribute grammar for evaluating simple expressions

<i>Production</i>	<i>Semantic rule</i>
$\langle E_1 \rangle \rightarrow \langle E_2 \rangle + \langle T \rangle$	$E_1.val := E_2.val + T.val$
$\langle E_1 \rangle \rightarrow \langle E_2 \rangle - \langle T \rangle$	$E_1.val := E_2.val - T.val$
$\langle E \rangle \rightarrow \langle T \rangle$	$E.val := T.val$
$\langle T_1 \rangle \rightarrow \langle T_2 \rangle * \langle F \rangle$	$T_1.val := T_2.val * F.val$
$\langle T_1 \rangle \rightarrow \langle T_2 \rangle / \langle F \rangle$	$T_1.val := T_2.val / F.val$
$\langle T \rangle \rightarrow \langle F \rangle$	$T.val := F.val$
$\langle F_1 \rangle \rightarrow - \langle F_2 \rangle$	$F_1.val := -F_2.val$
$\langle F \rangle \rightarrow (\langle E \rangle)$	$F.val := E.val$
$\langle F \rangle \rightarrow \text{unsigned_int}$	$F.val := \text{unsigned_int.val}$

- The *val* attribute of a (non)terminal holds the subtotal value of the subexpression described by the (non)terminal
- Nonterminals are *indexed* in the attribute grammar (e.g. $\langle T_1 \rangle$) to distinguish multiple occurrences of the nonterminal in a production

the parse tree by traversing the tree according to the *set and used* dependencies between attributes (an attribute must be set before it is read)

Production Action	Attribute flow
$\langle E \rangle \rightarrow \langle T \rangle \langle TT \rangle$ $TT.st := T.val$	
$\langle TT_1 \rangle \rightarrow + \langle T \rangle \langle TT_2 \rangle$ $TT_2.st := TT_1.st + T.val$	
$\langle TT \rangle \rightarrow e$ $TT.val := TT.st$	
$\langle TT_1 \rangle \rightarrow + \langle T \rangle \langle TT_2 \rangle$ $TT_1.val := TT_2.val$	



S- and L-Attributed Grammars

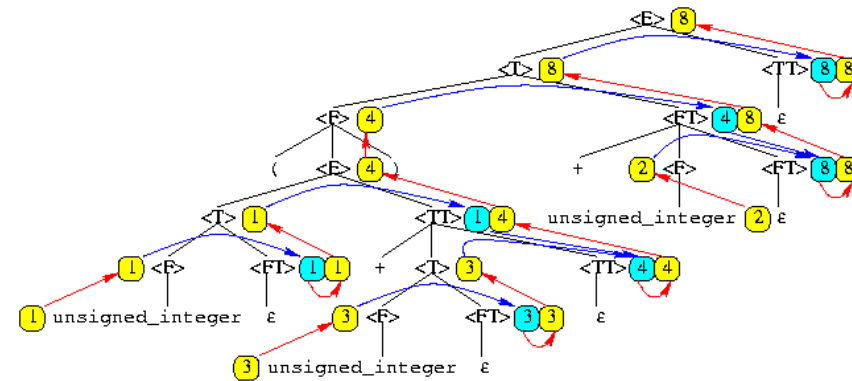
- A grammar is called *S-attributed* if *all* attributes are synthesized
- A grammar is called *L-attributed* if the parse tree traversal is *left-to-right* and *depth-first*
 - An essential grammar property for a *one-pass compiler*, because semantic rules can be applied directly during parsing and parse trees do not need to be kept in memory

Example L-attributed grammar for top-down parsing and evaluation of simple expressions

Production	Semantic rule
$\langle E \rangle \rightarrow \langle T \rangle \langle TT \rangle$	$TT.st := T.val; E.val := TT.val$
$\langle TT_1 \rangle \rightarrow + \langle T \rangle \langle TT_2 \rangle$	$TT_2.st := TT_1.st + T.val;$ $TT_1.val := TT_2.val$
$\langle TT_1 \rangle \rightarrow - \langle T \rangle \langle TT_2 \rangle$	$TT_2.st := TT_1.st - T.val;$ $TT_1.val := TT_2.val$
$\langle TT \rangle \rightarrow e$	$TT.val := TT.st$
$\langle T \rangle \rightarrow \langle F \rangle \langle FT \rangle$	$FT.st := F.val; T.val := FT.val$
$\langle FT_1 \rangle \rightarrow * \langle F \rangle \langle FT_2 \rangle$	$FT_2.st := FT_1.st * F.val;$ $FT_1.val := FT_2.val$
$\langle FT_1 \rangle \rightarrow / \langle F \rangle \langle FT_2 \rangle$	$FT_2.st := FT_1.st / F.val;$ $FT_1.val := FT_2.val$
$\langle FT \rangle \rightarrow e$	$FT.val := FT.st$
$\langle F_1 \rangle \rightarrow - \langle F_2 \rangle$	$F_1.val := -F_2.val$
$\langle F \rangle \rightarrow (\langle E \rangle)$	$F.val := E.val$
$\langle F \rangle \rightarrow \text{unsigned_int}$	$F.val := \text{unsigned_int.val}$

Example Decorated Parse Tree

- Fully decorated parse tree of $(1+3)*2$



Recursive Descent Parser for L-Attributed Grammar

- Productions for each nonterminal are implemented as a subroutine
- Subroutine returns synthesized attributes of the nonterminal
- Subroutine takes inherited attributes of the nonterminal as subroutine arguments

<i>Production</i>	<i>Semantic rule</i>
$\langle E \rangle \rightarrow \langle T \rangle \langle TT \rangle$ $\langle TT_1 \rangle \rightarrow + \langle T \rangle \langle TT_2 \rangle$ $\langle TT \rangle \rightarrow e$	$TT.st := T.val; E.val := TT.val$ $TT_2.st := TT_1.st + T.val;$ $TT_1.val := TT_2.val$ $TT.val := TT.st$
<pre>function E() Tval := T() TTst := Tval TTval := TT(TTst) return TTval function TT(TT1st) case (input_token()) of '+': Tval := T() TT2st := TT1st + Tval TT2val := TT(TT2st) TT1val := TT2val otherwise: TT1val := TT1st return TT1val</pre>	

Exercise: Write a recursive descent parser in Java to evaluate simple expressions. Answer: [▣](#)

Exercise: Write a recursive descent parser in Java to construct an abstract syntax tree (AST) for simple expressions. Modify the parser to generate Lisp expressions from the AST. Answer: [▣](#)