# 4. Syntax-Directed Translation

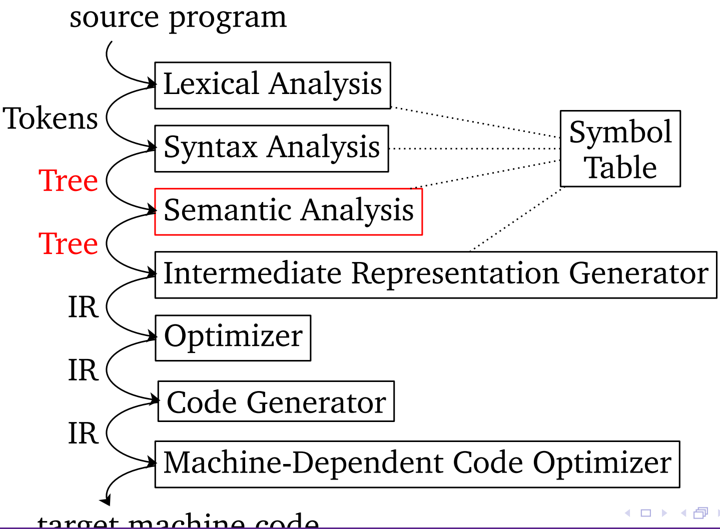
Eva Rose Kristoffer Rose

NYU Courant Institute

Compiler Construction (CSCI-GA.2130-001)
http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-4.pdf

September 25, 2014

# Context


source program
Lexical Analysis
Tokens
Syntax Analysis
Tree
Semantic Analysis
Tree
Intermediate Representation Generator
IR
Optimizer
IR
Code Generator
IR
Machine-Dependent Code Optimizer
target machine code
Symbol Table

# The Trees

| TREE | INTERIOR NODES | GRAMMAR |
|---|---|---|
| parse tree | nonterminals | concrete syntax |
| abstract syntax tree | programming constructs | abstract syntax |

## Example (abstract syntax)

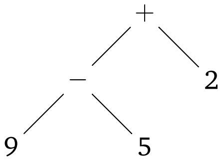



$$E \to E + E \mid E - E \mid \textbf{digit}$$

# The Trees

| TREE | INTERIOR NODES | GRAMMAR |
|---|---|---|
| parse tree | nonterminals | concrete syntax |
| abstract syntax tree | programming constructs | abstract syntax |

## Example (abstract syntax)



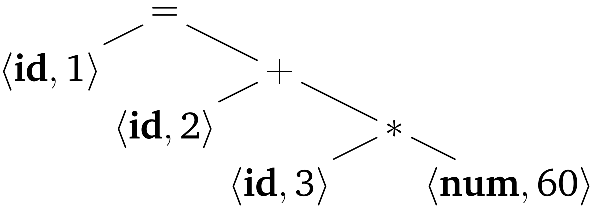

$$E \to E + E \mid E - E \mid \textbf{digit}$$

# Back to the introductory example...

# From Abstract Syntax Tree (AST)

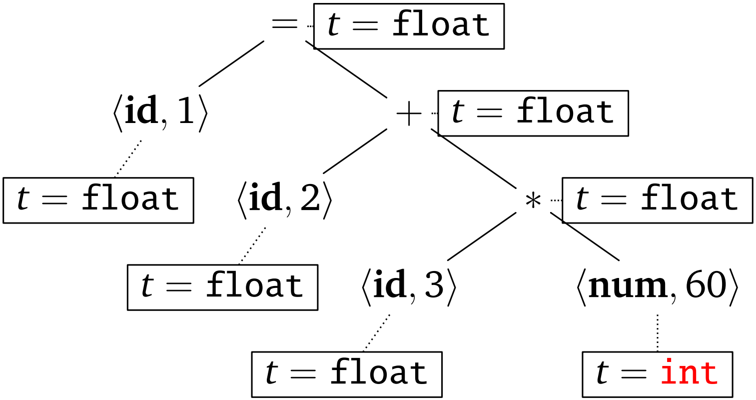
```
position = initial + rate * 60
```

parsed into abstract syntax tree:



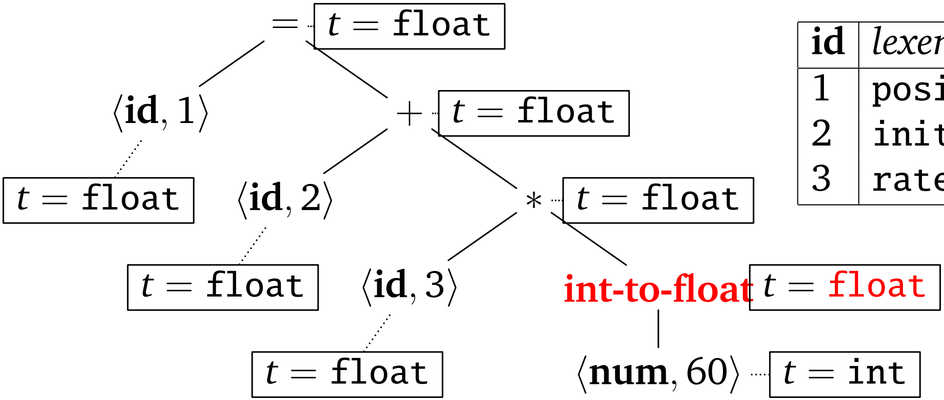

| **id** | *lexeme* |
|---|---|
| 1 | `position` |
| 2 | `initial` |
| 3 | `rate` |

# . . . to Annotated AST. . .




| **id** | *lexeme* | *t* |
|---|---|---|
| 1 | `position` | `float` |
| 2 | `initial` | `float` |
| 3 | `rate` | `float` |

# . . . to “Fixed” AST




| **id** | *lexeme* | *t* |
|---|---|---|
| 1 | `position` | `float` |
| 2 | `initial` | `float` |
| 3 | `rate` | `float` |

# Syntax-Directed Translation

We have built a parse (or AST) tree, now what? How will this tree and production rules help the translation?

Intuitively, we need to associate *something* with each production and each tree node:

- something that evaluates the meaning at each node,
- something that emits the meaning as program fragments.

# Syntax-Directed Translation

We have built a parse (or AST) tree, now what? How will this tree and production rules help the translation?

Intuitively, we need to associate *something* with each production and each tree node:

- something that evaluates the meaning at each node,
- something that emits the meaning as program fragments.

# Syntax-Directed Definition (SDD)

**Attributes** Each grammar symbol (terminal or nonterminal) has an *attribute* ("meaning") associated.

**Semantic Rules** Each production has *semantic rules* associated for computing the attributes.

# Example: SDD for Infix to Postfix Notation

Consider the grammar:

$$\begin{aligned} \mathit{expr} &\rightarrow \mathit{expr} + \mathit{term} \\ &\mid \mathit{expr} - \mathit{term} \\ &\mid \mathit{term} \\ \mathit{term} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

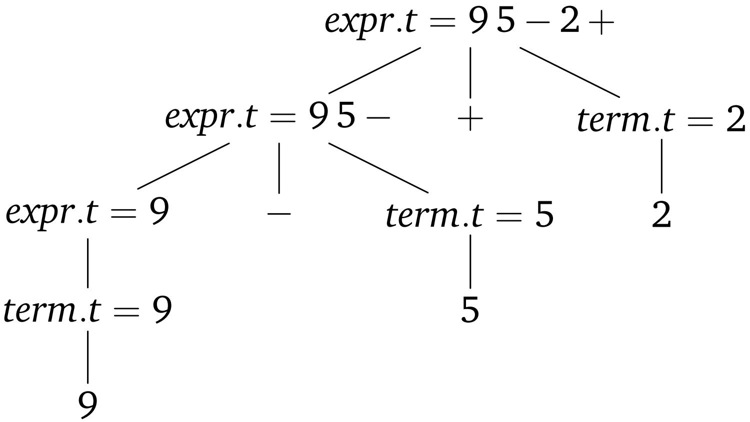
*expr* has an attribute $\mathit{expr}.t$, *term* has an attribute $\mathit{term}.t$.

# Example: SDD Infix to Postfix Notation

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $expr \rightarrow expr_1 + term_2$ | $expr.t = expr_1.t \mid\mid term_2.t \mid\mid '+'$ |
| $expr \rightarrow expr_1 - term_2$ | $expr.t = expr_1.t \mid\mid term_2.t \mid\mid '-'$ |
| $expr \rightarrow term_1$ | $expr.t = term_1.t$ |
| $term \rightarrow 0$ | $term.t =' 0'$ |
| $term \rightarrow 1$ | $term.t =' 1'$ |
| $\ldots$ | $\ldots$ |
| $term \rightarrow 9$ | $term.t =' 9'$ |

$\_ \mid\mid \_$ means concatenation, $\_{}_1$, $\_{}_2$ disambiguates

# Attribute Values in Parse Trees



# Translation Schemes

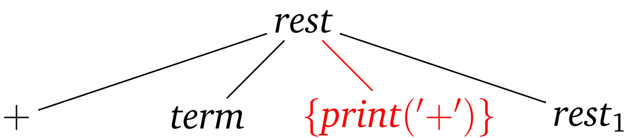
- Equivalent mechanism.
- Attaching program fragments to productions.
- The program fragments are called semantic actions/side-effects.
- They “emit” the program fragments during “tree traversal”.

**Example** ($rest \rightarrow +term$ $\{print('+')\}$ $rest_1$)

# Translation Schemes

- Equivalent mechanism.
- Attaching program fragments to productions.
- The program fragments are called semantic actions/side-effects.
- They "emit" the program fragments during "tree traversal".

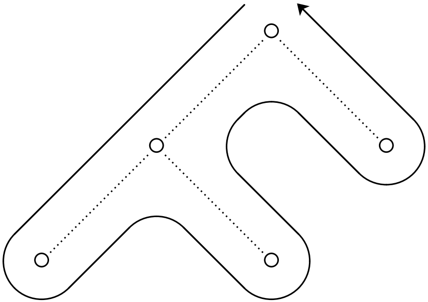
### Example ($rest \to +term\ \{print('+')\}\ rest_1$)



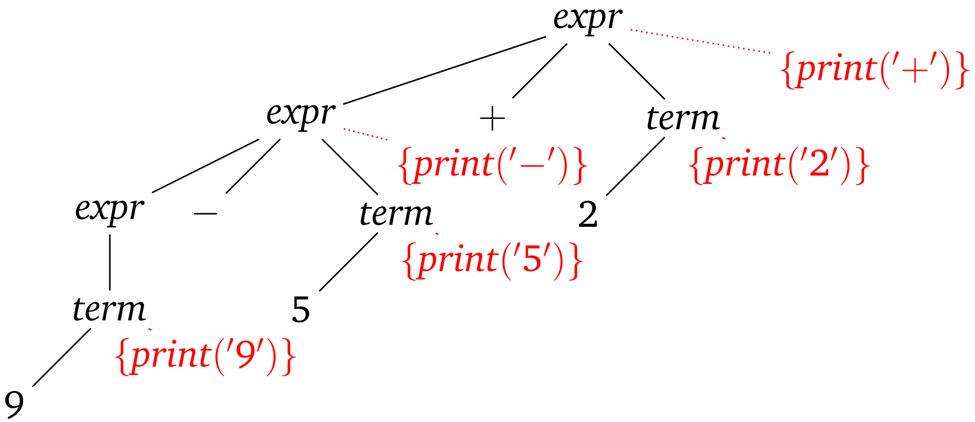
# Semantic Action Table: Infix to Postfix Notation

| PRODUCTIONS WITH SIDE-EFFECTS/ACTIONS |
|---|
| $expr \rightarrow expr_1 + term_2$ {print('+')} |
| $expr \rightarrow expr_1 - term_2$ {print('−')} |
| $expr \rightarrow term$ |
| $term \rightarrow 0$ {print('0')} |
| $term \rightarrow 1$ {print('1')} |
| $term \rightarrow 2$ {print('2')} |
| $term \rightarrow 3$ {print('3')} |
| $term \rightarrow 4$ {print('4')} |
| $term \rightarrow 5$ {print('5')} |
| $term \rightarrow 6$ {print('6')} |
| $term \rightarrow 7$ {print('7')} |
| $term \rightarrow 8$ {print('8')} |
| $term \rightarrow 9$ {print('9')} |

# Tree Traversal in Translation Schemes: Depth-First

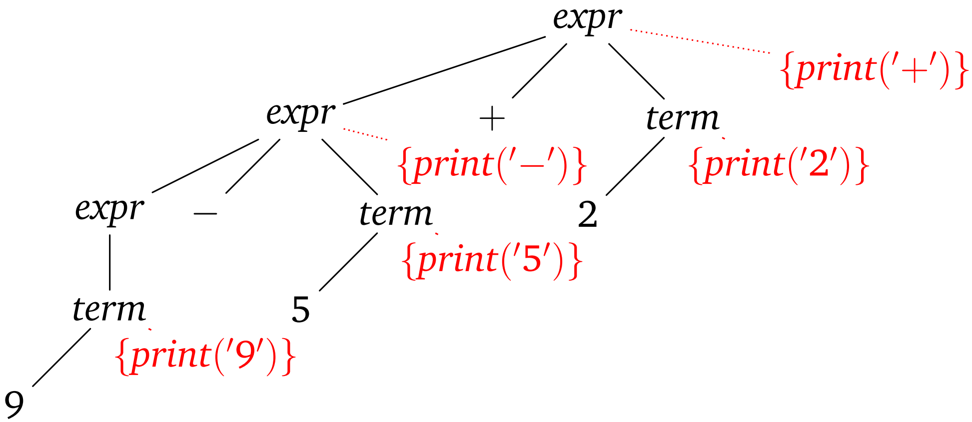


# Semantic Actions/Side-Effects in Parse Tree



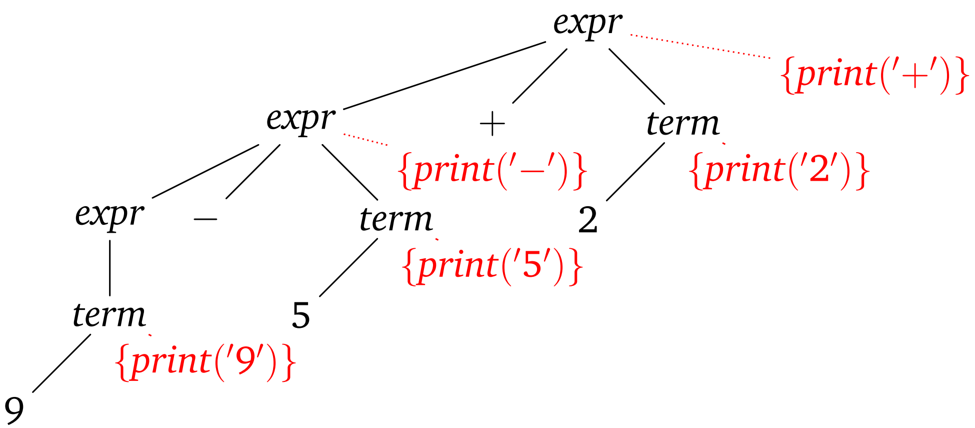

Input: $9 - 5 + 2$. Output (print): $9\,5 - 2\,+$. Single depth-first traversal

# Semantic Actions/Side-Effects in Parse Tree



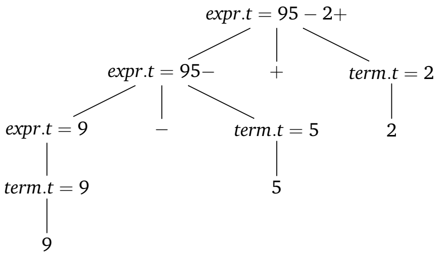

Input: $9 - 5 + 2$. Output (print): $9\,5 - 2\,+$. Single depth-first traversal

# Semantic Actions/Side-Effects in Parse Tree

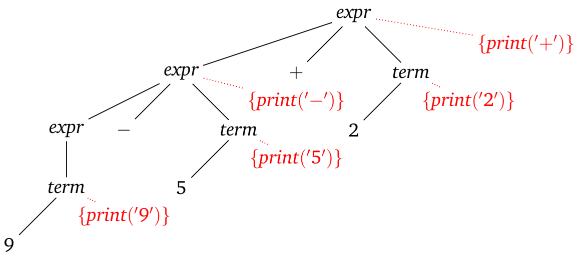



Input: $9 - 5 + 2$. Output (print): $9\,5 - 2\,+$. Single depth-first traversal

# Summary




attributes/semantic rules (HACS)




semantic actions/side-effects (yacc)

# Tree Traversal and Translation Schemes

When translation is defined in terms of semantic actions, the tree traversal order, that is the order in which the nodes are visited, becomes essential.

# Exercise

Construct an SDD that generates an attribute $t$ in *prefix notation* for each expression *expr*, for arithmetic expressions. (You only need to consider: '$+$','$-$', and '$*$' expressions). Prefix notation is where the operator comes before its operands; e.g., $-xy$ is the prefix notation for $x-y$.

# Solution: SDD for Infix Notation into Prefix Notation

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $expr \to expr_1 + term_2$ | $expr.t = '+' \,\|\|\, expr_1.t \,\|\|\, term_2.t$ |
| $expr \to expr_1 - term_2$ | $expr.t = '-' \,\|\|\, expr_1.t \,\|\|\, term_2.t$ |
| $expr \to expr_1 * term_2$ | $expr.t = '*' \,\|\|\, expr_1.t \,\|\|\, term_2.t$ |
| $expr \to term$ | $expr.t = term.t$ |
| $term \to 0$ | $term.t = '0'$ |
| $term \to 1$ | $term.t = '1'$ |
| $\ldots$ | $\ldots$ |
| $term \to 9$ | $term.t = '9'$ |

# Syntax-Directed Definition (SDD)

Recall the key definition:

**Grammar** Given a context-free grammar.

**Attributes** Each grammar symbol (terminal or nonterminal) has a set of *attributes* associated.

**Semantic Rules** Each production has a set of *semantic rules* associated for computing the attributes.

If $X$ is a symbol, $a$ is an attribute, then $X.a$ denotes *the value of* $a$ at node $X$.

# Synthesized and Inherited Attributes

**Synthesized** attributes at node N are defined only in terms of the attribute values of the children of N, and N itself.

**Inherited** attributes at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

# Example: SDD for Desk Calculator

| | Production | Semantic Rule |
|---|---|---|
| 1. | $L \rightarrow E\ \$$ | $L.val = E.val$ |
| 2. | $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| 3. | $E \rightarrow T$ | $E.val = T.val$ |
| 4. | $T \rightarrow T_1 * F$ | $T.val = T_1.val \times F.val$ |
| 5. | $T \rightarrow F$ | $T.val = F.val$ |
| 6. | $F \rightarrow (E)$ | $F.val = E.val$ |
| 7. | $F \rightarrow \textbf{digit}$ | $F.val = \textbf{digit}.lexval$ |

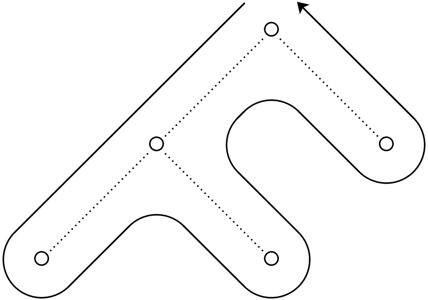
'*val*' and '*lexval*' are synthesized attributes.

# Characteristics of Desk Calculator

- S-attributed (only synthesized attributes)
- attribute grammar (without side-effects)

Order of attribute evaluation: any bottom-up traversal.

# Tree Traversal: Depth-First



Post-order (default). Pre-order. Binary trees: also in-order.

# Exercise: Desk Calculator

Give *annotated parse trees* for the following expressions:

- $(3+4)*(5+6)\ \$$
- $1*2*3*(4+5)\ \$$

(Parse trees showing attributes and their values)

# Exercise: Mingling with Inherited Attributes. . .

| | Production | Semantic Rules |
|---|---|---|
| 1. | $T \to F\, T'$ | $T'.inh = F.val;\; T.val = T'.syn$ |
| 2. | $T' \to *\, F\, T'_1$ | $T'_1.inh = T'.inh \times F.val;\; T'.syn = T'_1.syn$ |
| 3. | $T' \to \epsilon$ | $T'.syn = T'.inh$ |
| 4. | $F \to \textbf{digit}$ | $F.val = \textbf{digit}.lexval$ |

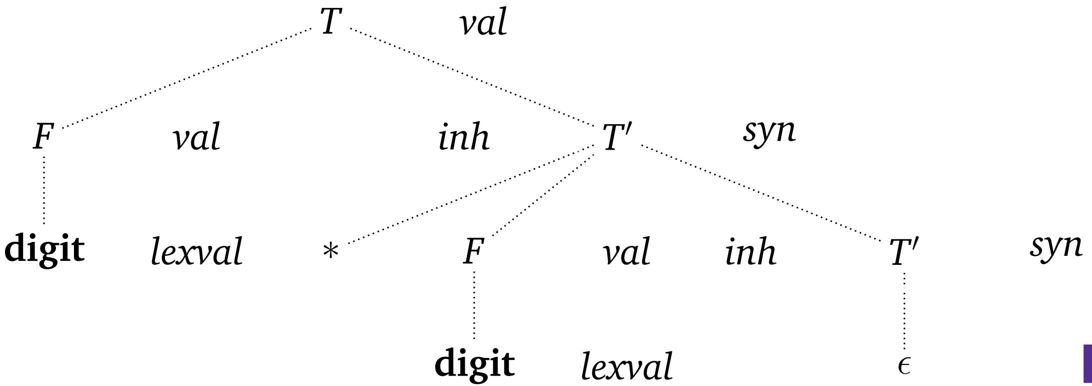
'*inh*' is inherited, and '*val*', '*syn*' are synthesized.

# Dependency Graphs

Dependency graphs depicts the flow of information among attributes.

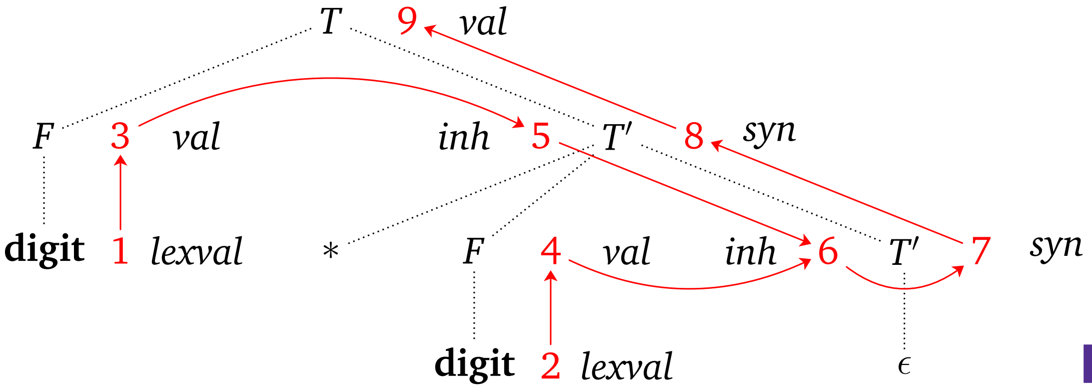
## Example: Attributes

Parse tree for **3** $*$ **5** based on SDD:



# Example: Attributes with Dependency Graph

Parse tree for **3** $*$ **5** based on SDD:



# Dependency Graphs

Central problem: Determining the efficient evaluation order for the attribute instances in a given parse tree.

# Calculating Attributes

Cycles possible!

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $A \to B$ | $A.s = B.i;\ B.i = A.s + 1$ |

When dependency graphs have cycles, evaluation is "stuck"!

# Well-behaved SDD Classes

Given an SDD, hard to tell if there exists a parse tree whose dependency graphs have cycles!

*Safe way to go:* use classes of SDDs that guarantee an evaluation order:

S-Attributed Definitions all attributes are synthesized.

L-Attributed Definitions "left" restrictions on dependency graph edges.

# Well-behaved SDD Classes

Given an SDD, hard to tell if there exists a parse tree whose dependency graphs have cycles!

*Safe way to go:* use classes of SDDs that guarantee an evaluation order:

S-Attributed Definitions all attributes are synthesized.

L-Attributed Definitions "left" restrictions on dependency graph edges.

# L-Attributed Definitions

Suppose an attribute rule for $A \to X_1 X_2 \ldots X_i \ldots X_n$ produces inherited $X_i.a$ attribute. The rule may only use:

- inherited attributes associated with the head ($A$),
- inherited or synthesized attributes associated with the occurences of symbols $X_1, X_2, \ldots, X_{i-1}$ located left of $X_i$.
- Inherited or synthesized attributes associated with the occurence of $X_i$ itself, as long as not part of a dependency graph cycle!

*or*, the produced attribute is synthesized.

# Example: Mingling with Inherited Attributes. . .

Decide if the SDD is L-attributed and argue :

| | Production | Semantic Rules |
|---|---|---|
| 1. | $T \to F\, T'$ | $T'.inh = F.val;\ T.val = T'.syn$ |
| 2. | $T' \to *\, F\, T'_1$ | $T'_1.inh = T'.inh \times F.val;\ T'.syn = T'_1.syn$ |
| 3. | $T' \to \epsilon$ | $T'.syn = T'.inh$ |
| 4. | $F \to \mathbf{digit}$ | $F.val = \mathbf{digit}.lexval$ |

‘*inh*’ is inherited, and ‘*val*’, ‘*syn*’ are synthesized.

# Example: Mingling with Inherited Attributes. . .

Decide if the SDD is L-attributed and argue :

| | Production | Semantic Rules |
|---|---|---|
| 1. | $T \rightarrow F\,T'$ | $T'.inh = F.val$; $T.val = T'.syn$ |
| 2. | $T' \rightarrow *\,F\,T'_1$ | $T'_1.inh = T'.inh \times F.val$; $T'.syn = T'_1.syn$ |
| 3. | $T' \rightarrow \epsilon$ | $T'.syn = T'.inh$ |
| 4. | $F \rightarrow \mathbf{digit}$ | $F.val = \mathbf{digit}.lexval$ |

'*inh*' is inherited, and '*val*', '*syn*' are synthesized.

# Example: Mingling with Inherited Attributes. . .

Decide if the SDD is L-attributed and argue :

| | Production | Semantic Rules |
|---|---|---|
| 1. | $T \to F\,T'$ | $T'.inh = F.val;\ T.val = T'.syn$ |
| 2. | $T' \to *\,F\,T'_1$ | $T'_1.inh = T'.inh \times F.val;\ T'.syn = T'_1.syn$ |
| 3. | $T' \to \epsilon$ | $T'.syn = T'.inh$ |
| 4. | $F \to \textbf{digit}$ | $F.val = \textbf{digit}.lexval$ |

'*inh*' is inherited, and '*val*', '*syn*' are synthesized.

# Another example

The following SDD is NOT L-attributed. Why?

| Production | Semantic Rules |
|---|---|
| $A \to B\,C$ | $A.s = B.b$ |
| | $B.i = F(C.c, A.s)$ |

where $B.i$ is inherited, and $A.s$, $C.c$ are synthesized.

# Another example

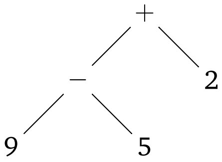
The following SDD is NOT L-attributed. Why?

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $A \rightarrow B\ C$ | $A.s = B.b$<br>$B.i = F(C.c, A.s)$ |

# Construction of Abstract Syntax Trees

| TREE | INTERIOR NODES | GRAMMAR |
|---|---|---|
| parse tree | nonterminals | concrete syntax |
| abstract syntax tree | programming constructs | abstract syntax |

## Example (abstract syntax)




$$E \rightarrow E + E \mid E - E \mid \textbf{digit}$$

# Construction of AST

## Example 1: S-attributed SDD for simple expresssions.

| | Production | Semantic Rule |
|---|---|---|
| 1. | $E \rightarrow E_1 + T_2$ | $E.node = \textbf{new}\ Node('+', E_1.node, T_2.node)$ |
| 2. | $E \rightarrow E_1 - T_2$ | $E.node = \textbf{new}\ Node('-', E_1.node, T_2.node)$ |
| 3. | $E \rightarrow T$ | $E.node = T.node$ |
| 4. | $T \rightarrow (\ E\ )$ | $T.node = E.node$ |
| 5. | $T \rightarrow \textbf{id}$ | $T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$ |
| 6. | $T \rightarrow \textbf{num}$ | $T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.entry)$ |

- Show the AST for $a - 4 + c$

# Construction of AST

## Example 2: L-attributed SDD for simple expresssions.

| | Production | Semantic Rule |
|---|---|---|
| 1. | $E \rightarrow T\,E'$ | $E.node = E'.syn;\ E'.inh = T.node$ |
| 2. | $E' \rightarrow +\,T\,E'_1$ | $E'_1.inh = \textbf{new}\ Node('+', E'.node, T.node)$<br>$E'.syn = E'_1.syn$ |
| 3. | $E' \rightarrow -\,T\,E'_1$ | $E'_1.inh = \textbf{new}\ Node('-', E'.node, T.node)$<br>$E'.syn = E'_1.syn$ |
| 4. | $E' \rightarrow \epsilon$ | $E'.syn = E'.inh$ |
| 5. | $T \rightarrow (\,E\,)$ | $T.node = E.node$ |
| 6. | $T \rightarrow \textbf{id}$ | $T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$ |
| 7. | $T \rightarrow \textbf{num}$ | $T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.entry)$ |

- Show AST and dependency graph for $a - 4 + c$

# Syntax-Directed Translation of Type Expressions

| | Production | Semantic Rule |
|---|---|---|
| 1 | $T \rightarrow B\, C$ | $T.syn = C.syn$ <br> $C.inh = B.syn$ |
| 2 | $B \rightarrow \textbf{int}$ | $B.syn = integer$ |
| 3 | $B \rightarrow \textbf{float}$ | $B.syn = integer$ |
| 4 | $C \rightarrow [\textbf{num}]C_1$ | $C.syn = array(\textbf{num}.val,\ C_1.syn)$ <br> $C_1.inh = C.inh$ |
| 5 | $C \rightarrow \epsilon$ | $C.syn = C.inh$ |

- Show AST and dependencies for **int**[2][3]

1. Introduction
2. SDT: Syntax-Directed Translation
3. SDD: Syntax-Directed Definition
4. HACS
   - Review
   - S-attributed SDDs
   - Recursive Translation Schemes

# HACS Review: Lexical Specification

```
space [ \t\n] ;

token Int   | ⟨Digit⟩+ ;
token Float | ⟨Int⟩ "." ⟨Int⟩ ;

token fragment Digit | [0–9] ;
```

# HACS Review: Parser Specification

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (\ E\ ) \mid \textbf{int} \mid \textbf{float}
\end{aligned}
\tag{4.1}
$$

$$
\begin{aligned}
\textbf{sort}\ \mathrm{Exp} \ &|\ [\![\langle \mathrm{Exp@1}\rangle + \langle \mathrm{Exp@2}\rangle]\!]\mathrm{@1} \\
&|\ [\![\langle \mathrm{Exp@2}\rangle * \langle \mathrm{Exp@3}\rangle]\!]\mathrm{@2} \\
&|\ [\![\langle \mathrm{Int}\rangle]\!]\mathrm{@3} \mid [\![\langle \mathrm{Float}\rangle]\!]\mathrm{@3} \\
&|\ \textbf{sugar}\ [\![(\langle \mathrm{Exp\#1@1}\rangle)]\!]\mathrm{@3} \to \mathrm{Exp\#1}\ ;
\end{aligned}
$$

$$
E \to E + E \mid E * E \mid \textbf{int} \mid \textbf{float}
$$

# HACS Review: Parser Specification

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to ( \, E \, ) \mid \textbf{int} \mid \textbf{float}
\end{aligned}
\tag{4.1}
$$

$$
\begin{aligned}
\text{sort Exp} \mid\ & [\![\langle \text{Exp@1} \rangle + \langle \text{Exp@2} \rangle]\!]\text{@1} \\
\mid\ & [\![\langle \text{Exp@2} \rangle * \langle \text{Exp@3} \rangle]\!]\text{@2} \\
\mid\ & [\![\langle \text{Int} \rangle]\!]\text{@3} \mid [\![\langle \text{Float} \rangle]\!]\text{@3} \\
\mid\ & \text{sugar } [\![(\langle \text{Exp\#1@1} \rangle)]\!]\text{@3} \to \text{Exp\#1} \ ;
\end{aligned}
$$

$$E \to E + E \mid E * E \mid \textbf{int} \mid \textbf{float}$$

# HACS Review: Parser Specification

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (\ E\ ) \mid \textbf{int} \mid \textbf{float}
\end{aligned}
\tag{4.1}
$$

$$
\begin{aligned}
\textbf{sort}\ \mathrm{Exp}\ &|\ [\![\langle \mathrm{Exp@1}\rangle + \langle \mathrm{Exp@2}\rangle]\!]\mathrm{@1} \\
&|\ [\![\langle \mathrm{Exp@2}\rangle * \langle \mathrm{Exp@3}\rangle]\!]\mathrm{@2} \\
&|\ [\![\langle \mathrm{Int}\rangle]\!]\mathrm{@3}\ |\ [\![\langle \mathrm{Float}\rangle]\!]\mathrm{@3} \\
&|\ \textbf{sugar}\ [\![(\langle \mathrm{Exp\#1@1}\rangle)]\!]\mathrm{@3} \to \mathrm{Exp\#1}\ ;
\end{aligned}
$$

$$E \to E + E \mid E * E \mid \textbf{int} \mid \textbf{float}$$

# This Week

- S-attributed SDDs.
- Recursive Translation Schemes.

# S-attributed SDD

Only synthesized attributes.

# Example: Type Synthesis SDD

| PRODUCTION | SEMANTIC RULES | |
|---|---|---|
| $E \to E_1 + E_2$ | $E.t = \text{Unif}(E_1.t, E_2.t)$ | (1) |
| $\mid E_1 * E_2$ | $E.t = \text{Unif}(E_1.t, E_2.t)$ | (2) |
| $\mid$ **int** | $E.t = \text{Int}$ | (3) |
| $\mid$ **float** | $E.t = \text{Float}$ | (4) |

# Example: Type Synthesis HACS Sorts

**sort** Type | Int | Float;

# Example: Type Synthesis HACS Unification Rules

```
sort Type | scheme Unif(Type,Type);
Unif(Int, Int) →Int;
Unif(Float, #) →Float;
Unif(#, Float) →Float;
```

# Example: Type Synthesis SDD Rule (1)

$$E \rightarrow E_1 + E_2 \quad \Big| \quad E.t = \mathrm{Unif}(E_1.t, E_2.t) \tag{1}$$

$$[\![\langle Exp\#1 \uparrow type(\#t1)\rangle + \langle Exp\#2 \uparrow type(\#t2)\rangle]\!] \uparrow type(Unif(\#t1, \#t2))$$

# Example: Type Synthesis SDD Rule (1)

$$E \to E_1 + E_2 \quad \Big| \quad E.t = \mathrm{Unif}(E_1.t, E_2.t) \tag{1}$$

$$[\![ \langle \mathit{Exp}\#1 \uparrow \mathit{type}(\#t1) \rangle + \langle \mathit{Exp}\#2 \uparrow \mathit{type}(\#t2) \rangle ]\!] \uparrow \mathit{type}(\mathit{Unif}(\#t1, \#t2))$$

# Example: Type Synthesis SDD Rule (1)

$$E \to E_1 + E_2 \quad \Big| \quad E.t = \mathrm{Unif}(E_1.t, E_2.t) \tag{1}$$

$$[\![\langle \mathit{Exp}\#1 \uparrow \mathit{type}(\#t1)\rangle + \langle \mathit{Exp}\#2 \uparrow \mathit{type}(\#t2)\rangle]\!] \uparrow \mathit{type}(\mathit{Unif}(\#t1, \#t2))$$

# Example: Type Synthesis SDD Rule (1)

$$E \to E_1 + E_2 \quad \Big| \quad E.t = \text{Unif}(E_1.t, E_2.t) \tag{1}$$

$$[\![\langle \mathit{Exp}\#1 \uparrow \mathit{type}(\#t1)\rangle + \langle \mathit{Exp}\#2 \uparrow \mathit{type}(\#t2)\rangle]\!] \uparrow \mathit{type}(\mathit{Unif}(\#t1, \#t2))$$

# Example: Type Synthesis SDD Rule (1)

$$E \to E_1 + E_2 \quad \Big| \quad E.t = \mathrm{Unif}(E_1.t, E_2.t) \tag{1}$$

$$[\![\langle \mathit{Exp}\#1 \uparrow \mathit{type}(\#t1)\rangle + \langle \mathit{Exp}\#2 \uparrow \mathit{type}(\#t2)\rangle]\!] \uparrow \mathit{type}(\mathit{Unif}(\#t1, \#t2))$$

# Example: Type Synthesis HACS

**attribute** $\uparrow$type(Type);

**sort** Exp; $\uparrow$type;

$[\![\ \langle \text{Exp\#1}\ \uparrow\text{type(\#t1)}\rangle + \langle \text{Exp\#2}\ \uparrow\text{type(\#t2)}\rangle\ ]\!]\uparrow\text{type(Unif(\#t1,\#t2))};$
$[\![\ \langle \text{Exp\#1}\ \uparrow\text{type(\#t1)}\rangle * \langle \text{Exp\#2}\ \uparrow\text{type(\#t2)}\rangle\ ]\!]\uparrow\text{type(Unif(\#t1,\#t2))};$
$[\![\ \langle \text{Int\#}\rangle\ ]\!]\uparrow\text{type(Int)};$
$[\![\ \langle \text{Float\#}\rangle\ ]\!]\uparrow\text{type(Float)};$

# Recursive Translation Scheme

// New syntax for value conversion from Int to Float:

$\textbf{sort}\ \mathrm{Exp}\ \mid\ [\![\mathtt{float}\ \langle \mathrm{Exp} \rangle]\!];$

# Recursive Translation Scheme (II)

```
// New scheme for inserting all needed int–to–float conversions:

sort Exp | scheme I2F(Exp);
```

# Recursive Translation Scheme (III)

// Cases for +:

$$\mathrm{I2F}(\llbracket\langle \mathrm{Exp}\#1{\uparrow}\mathrm{type}(\mathrm{Int})\rangle + \langle \mathrm{Exp}\#2{\uparrow}\mathrm{type}(\mathrm{Int})\rangle\rrbracket{\uparrow}\mathrm{type}(\#t))$$
$$\to \llbracket\langle \mathrm{Exp}\ \mathrm{I2F}(\#1)\rangle + \langle \mathrm{Exp}\ \mathrm{I2F}(\#2)\rangle\rrbracket{\uparrow}\mathrm{type}(\#t);$$

$$\mathrm{I2F}(\llbracket\langle \mathrm{Exp}\#1{\uparrow}\mathrm{type}(\mathrm{Float})\rangle + \langle \mathrm{Exp}\#2{\uparrow}\mathrm{type}(\mathrm{Int})\rangle\rrbracket{\uparrow}\mathrm{type}(\#t))$$
$$\to \llbracket\langle \mathrm{Exp}\ \mathrm{I2F}(\#1)\rangle + (\mathrm{float}\langle \mathrm{Exp}\ \mathrm{I2F}(\#2)\rangle)\rrbracket{\uparrow}\mathrm{type}(\#t);$$

$$\mathrm{I2F}(\llbracket\langle \mathrm{Exp}\#1{\uparrow}\mathrm{type}(\mathrm{Int})\rangle + \langle \mathrm{Exp}\#2{\uparrow}\mathrm{type}(\mathrm{Float})\rangle\rrbracket{\uparrow}\mathrm{type}(\#t))$$
$$\to \llbracket(\mathrm{float}\langle \mathrm{Exp}\ \mathrm{I2F}(\#1)\rangle) + \langle \mathrm{Exp}\ \mathrm{I2F}(\#2)\rangle\rrbracket{\uparrow}\mathrm{type}(\#t);$$

$$\mathrm{I2F}(\llbracket\langle \mathrm{Exp}\#1{\uparrow}\mathrm{type}(\mathrm{Float})\rangle + \langle \mathrm{Exp}\#2{\uparrow}\mathrm{type}(\mathrm{Float})\rangle\rrbracket{\uparrow}\mathrm{type}(\#t))$$
$$\to \llbracket\langle \mathrm{Exp}\ \mathrm{I2F}(\#1)\rangle + \langle \mathrm{Exp}\ \mathrm{I2F}(\#2)\rangle\rrbracket{\uparrow}\mathrm{type}(\#t);$$

# Recursive Translation Scheme (IV)

// Cases for $*$:

$$\mathrm{I2F}(\llbracket\langle \mathrm{Exp}\#1{\uparrow}\mathrm{type}(\mathrm{Int})\rangle * \langle \mathrm{Exp}\#2{\uparrow}\mathrm{type}(\mathrm{Int})\rangle\rrbracket{\uparrow}\mathrm{type}(\#t))$$
$$\to \llbracket\langle \mathrm{Exp}\ \mathrm{I2F}(\#1)\rangle * \langle \mathrm{Exp}\ \mathrm{I2F}(\#2)\rangle\rrbracket{\uparrow}\mathrm{type}(\#t);$$

$$\mathrm{I2F}(\llbracket\langle \mathrm{Exp}\#1{\uparrow}\mathrm{type}(\mathrm{Float})\rangle * \langle \mathrm{Exp}\#2{\uparrow}\mathrm{type}(\mathrm{Int})\rangle\rrbracket{\uparrow}\mathrm{type}(\#t))$$
$$\to \llbracket\langle \mathrm{Exp}\ \mathrm{I2F}(\#1)\rangle * (\mathrm{float}\langle \mathrm{Exp}\ \mathrm{I2F}(\#2)\rangle)\rrbracket{\uparrow}\mathrm{type}(\#t);$$

$$\mathrm{I2F}(\llbracket\langle \mathrm{Exp}\#1{\uparrow}\mathrm{type}(\mathrm{Int})\rangle * \langle \mathrm{Exp}\#2{\uparrow}\mathrm{type}(\mathrm{Float})\rangle\rrbracket{\uparrow}\mathrm{type}(\#t))$$
$$\to \llbracket(\mathrm{float}\langle \mathrm{Exp}\ \mathrm{I2F}(\#1)\rangle) * \langle \mathrm{Exp}\ \mathrm{I2F}(\#2)\rangle\rrbracket{\uparrow}\mathrm{type}(\#t);$$

$$\mathrm{I2F}(\llbracket\langle \mathrm{Exp}\#1{\uparrow}\mathrm{type}(\mathrm{Float})\rangle * \langle \mathrm{Exp}\#2{\uparrow}\mathrm{type}(\mathrm{Float})\rangle\rrbracket{\uparrow}\mathrm{type}(\#t))$$
$$\to \llbracket\langle \mathrm{Exp}\ \mathrm{I2F}(\#1)\rangle * \langle \mathrm{Exp}\ \mathrm{I2F}(\#2)\rangle\rrbracket{\uparrow}\mathrm{type}(\#t);$$

# Recursive Translation Scheme (II)

// Cases for literals:

$$\mathrm{I2F}(\llbracket\langle\mathrm{Int}\#1\rangle\rrbracket\uparrow\mathrm{type}(\#t)) \to \llbracket\langle\mathrm{Int}\#1\rangle\rrbracket\uparrow\mathrm{type}(\#t);$$
$$\mathrm{I2F}(\llbracket\langle\mathrm{Float}\#1\rangle\rrbracket\uparrow\mathrm{type}(\#t)) \to \llbracket\langle\mathrm{Float}\#1\rangle\rrbracket\uparrow\mathrm{type}(\#t);$$

*Questions?*

evarose@cs.nyu.edu krisrose@cs.nyu.edu