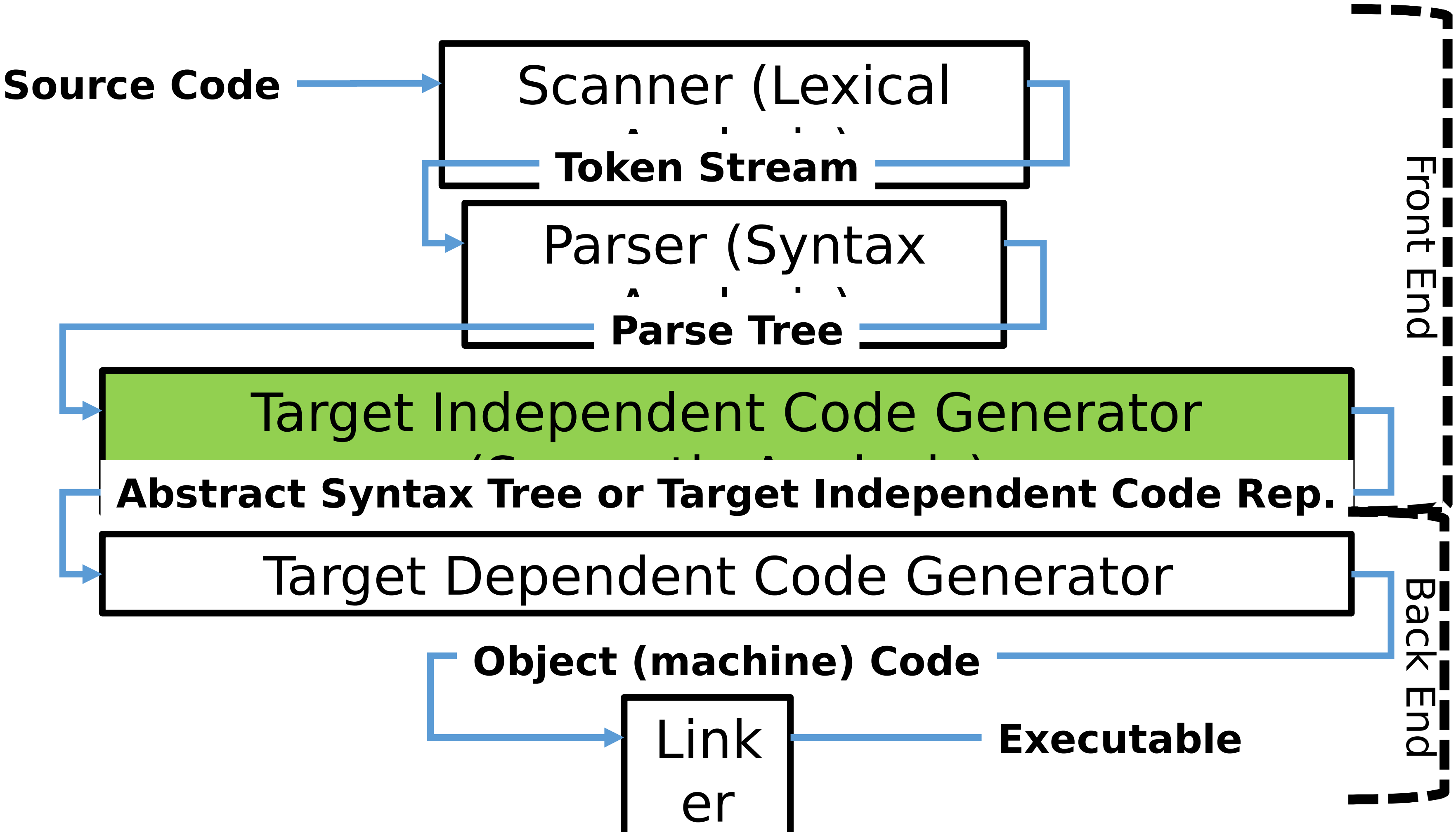
Semantic Analysis and Attribute Grammars

CSCI 3136: Principles of Programming Languages

Agenda

- S-Attributed and L-Attributed Grammars
- Examples
- Action Routines

Recall: Phases of Compilation



Example 1: L = $\{a^n b^n c^n | n \geq 0\}$

- This is not a context free language, but can be specified by an attribute grammar

CFG w/ Labeled Symbols	Semantic Rules
S → A_1 B_1 C_1	◃ **if** A_1.count != B_1.count **or** A_1.count != C_1.count, **error**
A → A_1 a	◃ A.count = A_1.count + 1
A → ε	◃ A.count = 0
B → B_1 b	◃ B.count = B_1.count + 1
B → ε	◃ B.count = 0
C → C_1 c	◃ C.count = C_1.count +
C → ε	◃ C.count = 0

Symbol	Attributes
A	count : int
B	count : int
C	count : int

- Example: Co

Example 2:
$L = \{\sigma \in \{a,b,c\}^*: |\sigma|_a = |\sigma|_b = |\sigma|_c\}$

CFG w/ Labeled Symbols	**Semantic Rules**
$S \rightarrow X_1$	◁ **if** X_1.aCount != X_1.bCount **or** X_1.aCount != X_1.cCount, **error**
$X \rightarrow a\ X_1$	◁ X.aCount = X_1.aCount + 1; X.bCount = X_1.bCount; X.cCount = X_1.cCount;
$X \rightarrow b\ X_1$	◁ X.bCount = X_1.bCount + 1; X.aCount = X_1.aCount; X.cCount = X_1.cCount;
$X \rightarrow c\ X_1$	◁ X.cCount = X_1.cCount + 1; X.bCount = X_1.bCount; X.aCount = X_1.aCount;
$X \rightarrow \varepsilon$	◁ X.aCount = 0; X.bCount = 0; X.cCount = 0;

Symbol	**Attributes**
X	aCount : int bCount : int cCount : int

Why do we need the **S → X** production?

Types of Attributes

- The previous examples are of *synthesized* (bottom up) attribute grammars.
- There are two types of Attributes
  - ***Synthesized* attributes** are computed in the RHS and stored in LHS
  - ***Inherited* attributes** are computed using LHS and RHS and used by symbols further to the right.

Example 3: L = $\{a^n b^n c^n | n \geq 0\}$

- Using inherited attributes instead of synthesized.

CFG w/ Labeled Symbols	Semantic Rules
$S \rightarrow A_1\ B_1\ C_1$	◃ B_1.iCount = A_1.count; C_1.iCount = A.count
$A \rightarrow A_1$ a	◃ A.count = A_1.count + 1
$A \rightarrow \varepsilon$	◃ A.count = 0
$B \rightarrow B_1$ b	◃ B_1.iCount = B.iCount -
$B \rightarrow \varepsilon$	◃ **if** B.iCount != 0, **error**
$C \rightarrow C_1$ c	◃ C_1.iCount = C.iCount
$C \rightarrow \varepsilon$	◃ **if** C.iCount != 0, **erro**

Symbol	Attributes
A	count : int
B	**iCount : int**
C	**iCount : int**

- Example: Consider parsing: aa

Example 4: Using Inherited Attributes
$L = \{\sigma \in \{a,b,c\}^*: |\sigma|_a \models |\square|_b \models |\square|_c\}$

CFG w/ Labeled Symbols	Semantic Rules
$S \rightarrow X_1$	◁ X_1.aCount = 0; X_1.bCount = 0; X_1.cCount = 0;
$X \rightarrow a\ X_1$	◁ X_1.aCount = X.aCount + 1; X_1.bCount = X.bCount; X_1.cCount = X.cCount;
$X \rightarrow b\ X_1$	◁ X_1.bCount = X.bCount + 1; X_1.aCount = X.aCount; X_1.cCount = X.cCount;
$X \rightarrow c\ X_1$	◁ X_1.cCount = X.cCount + 1; X_1.bCount = X.bCount; X_1.aCount = X.aCount;
$X \rightarrow \varepsilon$	◁ **if** X.aCount != X.bCount **or** X.aCount != X.cCount , **error**

Symbol	Attributes
X	aCount : int bCount : int cCount : int

Recap

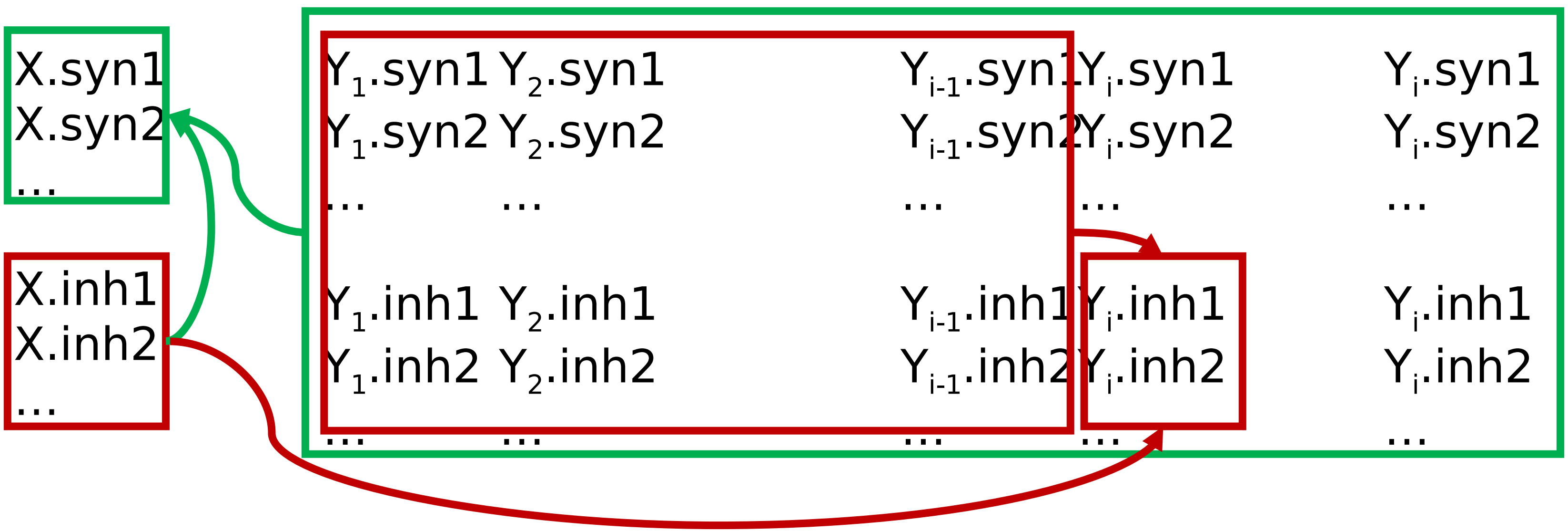
- Parse trees can be annotated or decorated with attributes and rules, which are executed as the tree is traversed.
- Synthesized attributes
  - Attributes of LHS of production are computed from attributes of RHS
  - Attributes flow bottom-up in the parse tree.
- Inherited attributes
  - Attributes in RHS are computed from attributes of LHS and symbols in RHS preceding them.
  - Attributes flow top-down in the parse tree.

S-Attributed and L-Attributed Grammars

- S-attributed grammar
  - All attributes are synthesized.
  - Attributes flow bottom-up.
- L-attributed grammar
  - Variables have both inherited and synthetic attributes
  - For each production $X \rightarrow Y_1 Y_2 \ldots Y_k$,
    - X.syn depends on
      - X.inh
      - Y_1.inh, Y_1.syn, Y_2.inh, Y_2.syn, . . . Y_k.inh, Y_k.syn
  - For all $1 \leq i \leq k$, Y_i.inh depends on
    - X.inh
    - Y_1.inh, Y_1.syn, Y_2.inh, Y_2.syn, . . . , Y_{i-1}.inh, Y_{i-1}.syn
- S-attributed grammars are a special case of L-attributed grammars.

Data Flow in L-Attributed Grammars

$X \rightarrow Y_1 Y_2 \; . \; . \; . \; Y_{i-1} Y_i \ldots Y$



Computing L-Attributed Grammars

```
execute_rules( Node t, Node [] left_sibs ):
  # Don't use t.synthetic and t.parent.synthetic
  t.compute_inherited( t.parent, left_sibs )

  children = []
  for each child of t:
    execute_rules( child, children )
    children.add( child )

  # Don't use t.synthetic and t.parent.synthetic
  t.compute_synthetic( children )
  return
```

Motivation: Why are they useful?

- In many cases context free grammars that capture associativity rules are not LL(1)
- We can rewrite the grammars to be LL(1) but…
- Resulting grammars do no capture associativity rules
- So, use attribute (L-attributed) grammars to capture the associativity rules.

Example: Left Associative Grammar

- Grammar
  - $E \to E\ A\ T$
  - $E \to T$
  - $T \to$ Int
  - $A \to +$
  - $A \to -$
- Parsing the expression

  $5 - 2 + 3$ illlustrates left associativity: $(5 - 2) + 3$
- This grammar is not LL(1)

Predictor Table	
Production	**Predictor Set**
$E \to EAT$	{Int}
$E \to T$	{Int}
$T \to$ Int	{Int}
$A \to +$	{+}
$A \to$ -	{-}

Example: Refactored Grammar

- Grammar
  - E → T E’
  - E’ → ε
  - E’ → A T E’
  - T→ Int
  - A → +
  - A → −
- Parsing the expression
  5 − 2 + 3 illlustrates wrong associativity: 5 − (2 + 3)
- This grammar is LL(1)

Predictor Table	
Production	**Predictor Set**
E → T E’	{Int}
E’ → ε	{ε}
E’ → A T E’	(+, -}
T→ Int	{Int}
A → +	{+}
A→ -	{-}

Use an L-Attributed Grammar to Fix Left Associat

Sym	Attributes
E	val : int
E’	val : int **op : int**
T	val : int
A	func : operation
Int	val : String

Idea: Carry forward the left most computed value to ensure left associativity

• Try par

Labeled CFG	Semantic Rules
E → T E'_1	◁ E_1′.op = T_1.val; E.val = E′.val
E’ → ε	◁ E′.val = E′.op
E’ → A_1 T_1 E'_1	◁ E_1′.op = A.func(E′.op, T_1.val); E′.val = E_1′.val
T→ Int_1	◁ T.val = Str2Int(Int_1.val)
A → +	◁ A.func = add
A→ -	◁ A.func = add

Example: Error Checking

Labeled CFG	Semantic Rules
Assignment → LValue$_1$ ‘=‘ Expr$_1$	◁ **if not** assignable(Lvalue$_1$.t, Expr$_1$.t), **error**
LValue → Id$_1$ ArrIdx$_1$	◁ **if not** declared(Id$_1$.name), **error** ◁ **if not** indexable(Id$_1$.name, ArrIdx$_1$.dim), **error**
ArrIdx → ε	◁ ArrIdx.dim = 0
ArrIdx → ‘[‘ Expr$_1$ ‘]’	**not** isType(Expr$_1$.t, Integer), **error** rrIdx.dim = ArrIdx$_1$.dim + 1

Sym	Attributes
Assignment	
LValue	t : Type
Id	name : String
ArrIdx	dim : int
Expr	t : Type

Example: Generate Java Code

Labeled CFG	Semantic Rules
$E \rightarrow E_1 A_1 T_1$	◃ E.tmp = tmpSeqNum++ **output(** “int tmp%d = tmp%d %s %s;”, E.tmp, E_1.tmp, A_1.op, T_1.var **)**
$E \rightarrow T_1$	◃ E.tmp = tmpSeqNum++ **output(** “int tmp%d = %s;”, E.tmp, T_1.var **)**
$T \rightarrow Id_1$	◃ T.var = id_1.name
$A \rightarrow$ ‘+’	◃ A.op = “+”
$A \rightarrow$ ‘-’	◃ A.op = “-”

Sym	Attributes
E	tmp : int
T	var : String
Id	name : String
A	op : String

Try generating Java code for the expression:
a + b - c

Action Routines

- Action routines are instructions for ad-hoc translation interleaved with parsing
- Parser generators allow programmers to specify action routines as part of the grammar
- Action routines can appear anywhere in a rule (as long as the grammar is LL(1)).
- Example
  - $E_1 \rightarrow A\ T$ **$\{E_2.op = A.fun(E_1.op, T.val)\}$** E_2 **$\{E_1.val = E_2.val\}$**
- Action routines are supported, for example, in yacc and bison