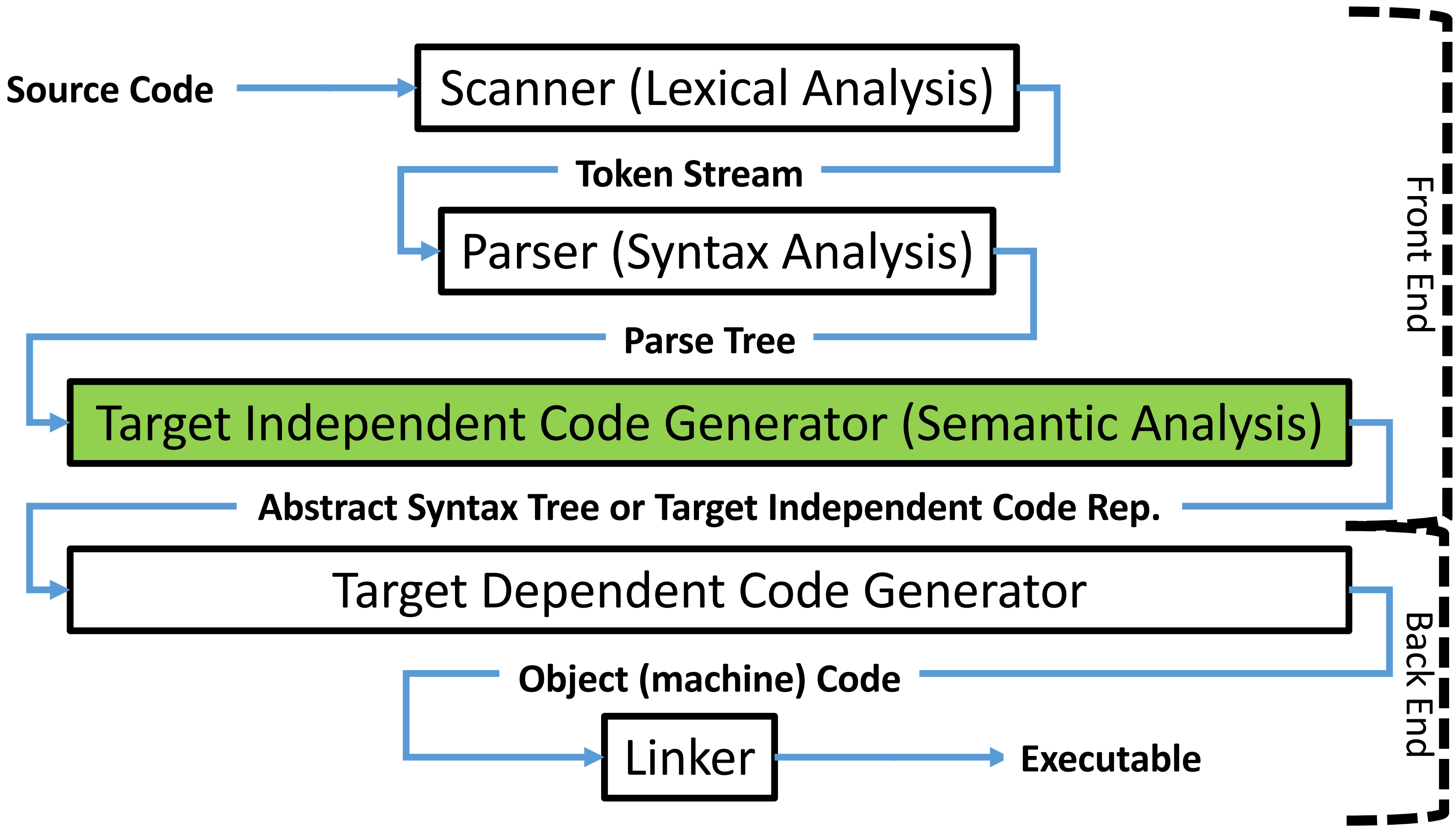
Semantic Analysis and Attribute Grammars

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - Assignment 5 is out and due June 28
- Readings: Read Chapter 4
- Lecture Contents
  - Motivation for Semantic Analysis
  - Semantic Analysis
  - Semantic Rules
  - Attribute Grammars
  - S-Attributed and L-Attributed Grammars

Recall: Phases of Compilation



Motivation

- Syntax
  - Describes form of a valid program
  - Can be described by a context-free grammar
- Semantics
  - Describes meaning of a program
  - Cannot be be described by a context-free grammar
- Some syntactic constraints may be enforced by semantic analysis

  E.g., order of evaluation of expressions

The Semantic Analysis Phase

- Use the syntax generated tree (from parser)
- Enforce semantic rules
- Build intermediate representation
  - e.g., abstract syntax tree
- Populate symbol table
- Pass results to intermediate code generator

Representation and Implementation

- Two approaches
  - Interleaved with syntactic analysis
  - As a separate phase
- Formal representation: Attributes grammars
- Observation:
  - Syntax grammars specify syntactic rules
  - Attribute grammars specify semantic rules

  The role of each phase is to enforce the corresponding rules.

Semantic Rules

- Two types: *static* and *dynamic*
- Static semantic rules
  - Enforced by compiler at compile time
  - Example: Do not use undeclared variable
- Dynamic semantic rules
  - Compiler generates code for enforcement at run time
  - Examples: division by zero, array index out of bounds
- Some compilers allow these checks to be disabled
  - To make code more efficient

Attribute Grammars

- Warning: When discussing attribute grammars we use the term ***symbol*** to mean variable, nonterminal, or terminal.
- Definition: An *attribute grammar* is an augmented context free grammar
  - Symbols are augmented with 0 or more attributes
    - Attributes are fields that store state or data
  - Productions are augmented with semantic rules (operations)
- Semantic rules
  - Copy attribute values between symbols
  - Evaluate attribute values using semantic functions
  - Enforce constraints on attribute values
  - Generate errors or warnings

Intuition

- Each node in a parse tree corresponds to a symbol
- The fields associated with a symbol are stored in the nodes of the parse tree
- Semantic rules refer to how we manipulate the fields stored in the nodes of the parse tree
- Think of various tree traversals from CSCI 2110. Semantic rules are implemented by performing traversals on the tree and doing some computation at each node.

Computing on the Parse Tree

- **Key Idea: Attribute grammars specify a computation on the parse tree**
- Examples of computations:
    - Symbol table generation
    - Type checking
    - Expression evaluation
    - Extended syntax checking
    - Code generation
    - **Code execution (in an interpreter)!**

Example of an Attribute Grammar (Expression Evaluation)

CFG with Labeled Symbols	Semantic Rules
$S \rightarrow + S_1\ S_2$	◃ $S.val = S_1.val + S_2.val$
$S \rightarrow - S_1\ S_2$	◃ $S.val = S_1.val - S_2.val$
$S \rightarrow * S_1\ S_2$	◃ $S.val = S_1.val * S_2.val$
$S \rightarrow / S_1\ S_2$	◃ $S.val = S_1.val / S_2.val$
$S \rightarrow$ neg S_1	◃ $S.val = - S_1.val$
$S \rightarrow Integer_1$	◃ $S.val = int(Integer_1.val)$

Symbol	Attributes
S	val : int
Integer	val : String

- Idea: We can apply semantic rules directly to our parse tree.

  E.g. + - 1 2 * 3 4

Example 2: $L = \{a^n b^n c^n \mid n \geq 0\}$ (Extended Syntax Checking)

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

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

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

- Example: Consider parsing: aaaabbbbcccc

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: $L = \{a^n b^n c^n \mid n \geq 0\}$

- 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_1.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 - 1
$B \rightarrow \varepsilon$	**if** B.iCount != 0, **error**
$C \rightarrow C_1\ c$	C_1.iCount = C.iCount - 1
$C \rightarrow \varepsilon$	**if** C.iCount != 0, **error**

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

synthetic

inherited

- Example: Consider parsing: aaaabbbbcccc

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.