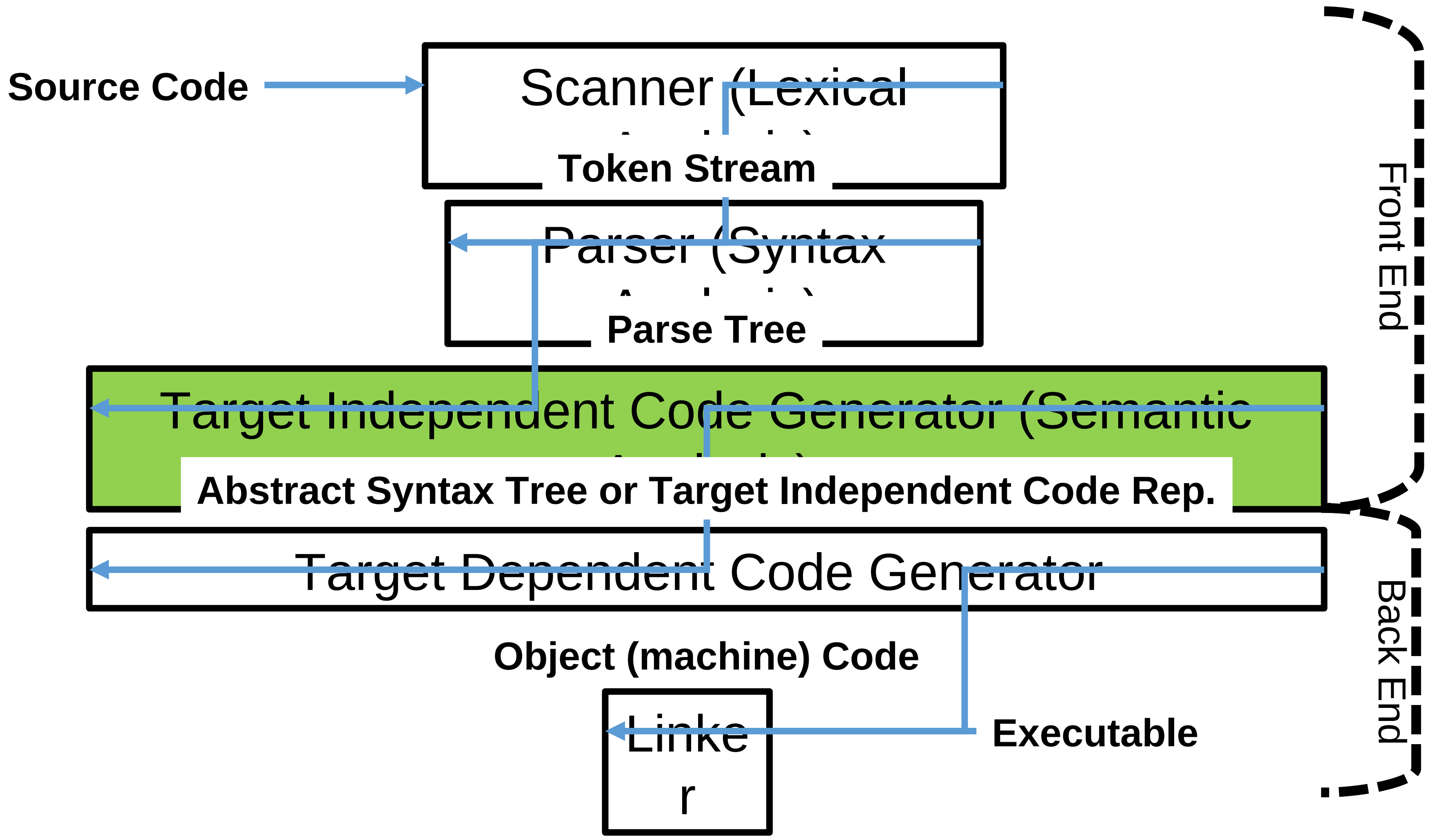
Semantic Analysis and Attribute Grammars

CSCI 3136: Principles of Programming Languages

Agenda

- Motivation
- Semantic Analysis
- Semantic Rules
- Attribute Grammars
- S-Attributed and L-Attributed Grammars

Recall: Phases of Compilation



Source Code
Scanner (Lexical Analysis)
Token Stream
Parser (Syntax Analysis)
Parse Tree
Target Independent Code Generator (Semantic Analysis)
Abstract Syntax Tree or Target Independent Code Rep.
Target Dependent Code Generator
Object (machine) Code
Linker
Executable
Front End
Back End

Our Compiler So Far ...

- Lexical Analyzer: A DFSA that breaks the program down into tokens (tokens are described using a regular language) – the lexical structure of the programming language
- Parser: An LL(1) or LR(1) recogniser for a context free language that builds a parse tree or abstract syntax tree – the syntactic structure of the programming language

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 are enforced by semantic analysis

  E.g., use of identifier only after its declaration

The Semantic Analysis Phase

- Use the syntax generated tree (from parser)
- Enforce semantic rules
- Potentially simplify the parse tree into an abstract syntax tree (AST)
- Populate symbol table (though this is usually done during parsing)
- Do any needed overload resolution
- Check for detectable errors (e.g., array bounds with constants – not all compilers do this)
- Pass results to intermediate code generator

Representation and Implementation

- Two approaches
  - Interleaved with syntactic analysis (normal way)
  - As a separate phase (easier to program)
- Formal representation: Attribute 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

Attribute Grammars

- Definition: An *attribute grammar* is an augmented context free grammar
  - Symbols are augmented with 0 or more attributes
    - Attributes are variables 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

Example of an Attribute Grammar

CFG with Labeled Symbols	Semantic Rules
S → + S1 S2	□ S.val = S1.val + S2.val
S → − S1 S2	□ S.val = S1.val - S2.val
S → X S1 S2	□ S.val = S1.val * S2.val
S → / S1 S2	□ S.val = S1.val / S2.val
S → neg S1	□ S.val = - S1.val
S → Integer1	□ S.val = S → Int1

Symbol	Attributes
S	val : int
Integer	val : String

- pply semantic rules directly to our parse tree.

E.g. + - 1 2 * 3 4

Example 2: L = {anbncn|n ≥ 0}

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

CFG w/ Labeled Symbols	Semantic Rules
S → A1 B1 C1	☐ **if** A1.count != B1.count **or** A1.count != C1.count then **error**
A → A1 a	☐ A.count = A1.count + 1
A → ε	☐ A.count = 0
B → B1 b	☐ B.count = B1.count + 1
B → ε	☐ B.count = 0
C → C1 c	☐ C.count = C1.count + 1
C → ε	☐ 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 = {anbncn|n ≥ 0}

- Using inherited attributes instead of synthesized.

CFG w/ Labeled Symbols	Semantic Rules
S → A1 B1 C1	☐ B1.iCount = A1.count; C1.iCount = A1.count
A → A1 a	☐ A1.count = A.count + 1
A → ε	☐ A.count = 0
B → B1 b	☐ B1.iCount = B.iCount - 1
B → ε	☐ **if** B.iCount != 0, **error**
C → C1 c	☐ C1.iCount = C.iCount - 1
C → ε	☐ **if** C.iCount != 0, **error**

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

- 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.