Compiler Design

Lecture-7

Introduction to Syntax analysis

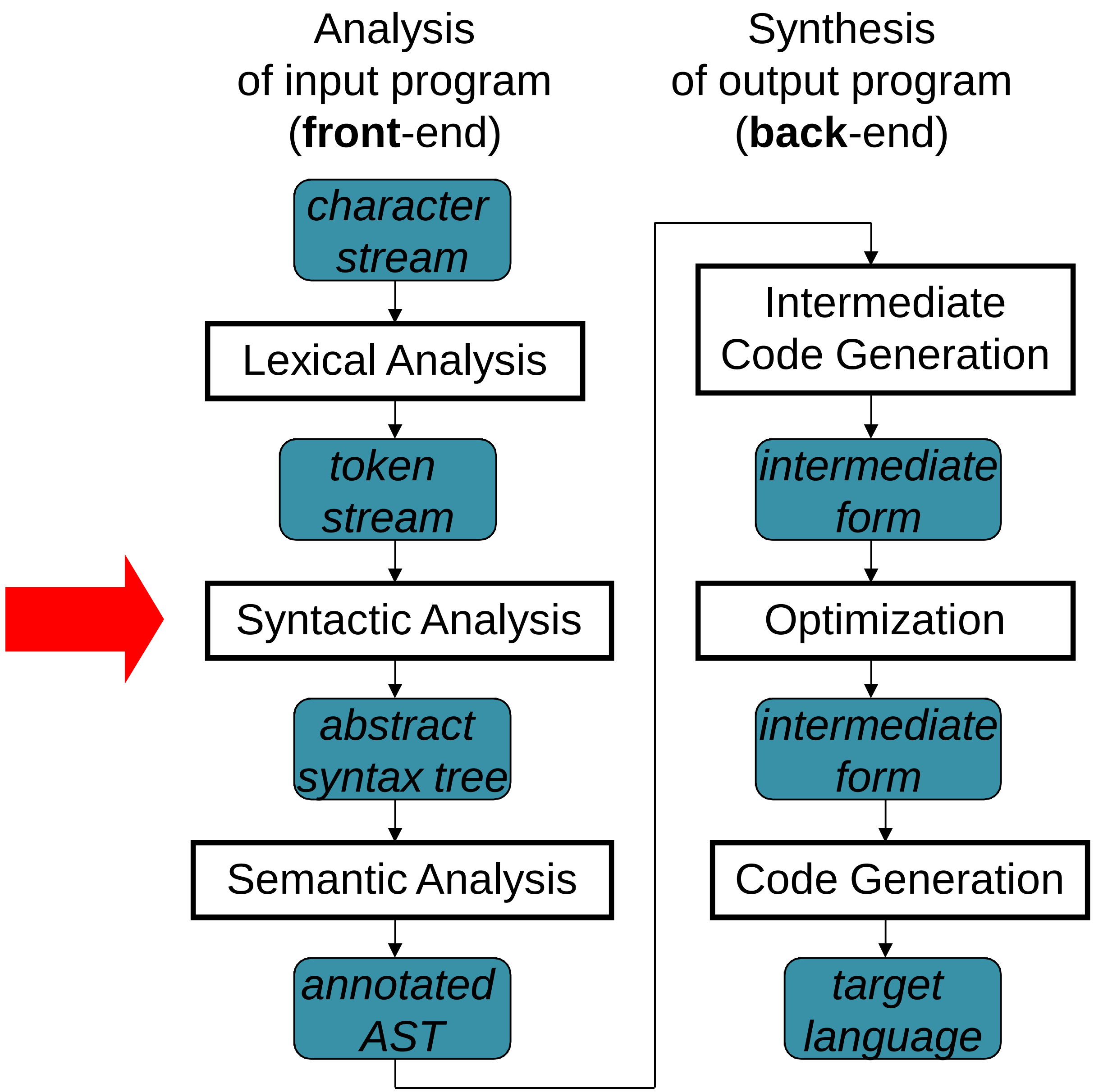
Topics Covered

- Syntax analysis
- CFG

Syntactic Analysis

Syntactic analysis, or parsing, is the second phase of compilation: The token file is converted to an abstract syntax tree.

Compiler Passes



Syntactic Analysis / Parsing

- Goal: Convert token stream to **abstract syntax tree**
- Abstract syntax tree (AST):
  - Captures the structural features of the program
  - Primary data structure for remainder of compilation
- Three Part Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing / building ASTs
  - Study the miniJava Implementation

Context-free Grammars

- Compromise between
  - Res, can't nest or specify recursive structure
  - General grammars, too powerful, undecidable
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - Easy to parse; but also allow restrictions for speed
- Not perfect
  - Cannot capture semantics, as in, "variable must be declared," requiring later semantic pass
  - Can be ambiguous
- EBNF, Extended Backus Naur Form, is popular notation

CFG Terminology

- **Terminals** -- alphabet of language defined by CFG
- **Nonterminals** -- symbols defined in terms of terminals and nonterminals
- **Productions** -- rules for how a nonterminal (lhs) is defined in terms of a (possibly empty) sequence of terminals and nonterminals
  - Recursion is allowed!
- Multiple productions allowed for a nonterminal, **alternatives**
- State symbol -- root of the defining language

```
Program ::= Stmt
Stmt ::= if ( Expr ) then Stmt else Stmt
Stmt ::= while ( Expr ) do Stmt
```

EBNF Syntax of initial MiniJava

```
Program       ::= MainClassDecl { ClassDecl }
MainClassDecl ::= class ID {
                  public static void main
                  ( String [ ] ID ) { { Stmt } }}
ClassDecl     ::= class ID [ extends ID ] {
                  { ClassVarDecl } { MethodDecl }
  }
ClassVarDecl  ::= Type ID ;
MethodDecl    ::= public Type ID
                  ( [ Formal { , Formal } ] )
                  { { Stmt } return Expr ; }
Formal        ::= Type ID
Type          ::= int |boolean | ID
```

Initial miniJava [continued]

```
Stmt ::= Type ID ;
       | { {Stmt} }
       | if ( Expr ) Stmt else Stmt
       | while ( Expr ) Stmt
       | System.out.println ( Expr ) ;
       | ID = Expr ;
Expr ::= Expr Op Expr
       | ! Expr
       | Expr . ID( [ Expr { , Expr } ] )
       | ID | this
       | Integer | true | false
       | ( Expr )
Op   ::= + | - | * | /
       | < | <= | >= | > | == | != | &&
```

RE Specification of initial MiniJava Lex

```
Program ::= (Token | Whitespace)*
Token ::= ID | Integer | ReservedWord | Operator |
           Delimiter
ID ::= Letter (Letter | Digit)*
Letter ::= a | ... | z | A | ... | Z
Digit ::= 0 | ... | 9
Integer ::= Digit+
ReservedWord::= class | public | static | extends |
        void | int | boolean | if | else |
       while|return|true|false| this | new | String
       | main | System.out.println
Operator ::= + | - | * | / | < | <= | >= | > | == |
       != | && | !
Delimiter ::= ; | . | , | = | ( | ) | { | } | [ | ]
```

Derivations and Parse Trees

Derivation: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals

Parsing: inverse of derivation

- Given a sequence of terminals (a\k\a tokens) want to recover the nonterminals representing structure

Can represent derivation as a **parse tree**, that is, the **concrete** syntax tree

Example Grammar

```
E  ::= E op E | - E | ( E ) | id
op ::= + | - | * | /
```

a * (b + - c)