Organization of Programming Languages CS 3200/5200D

Lecture 03

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

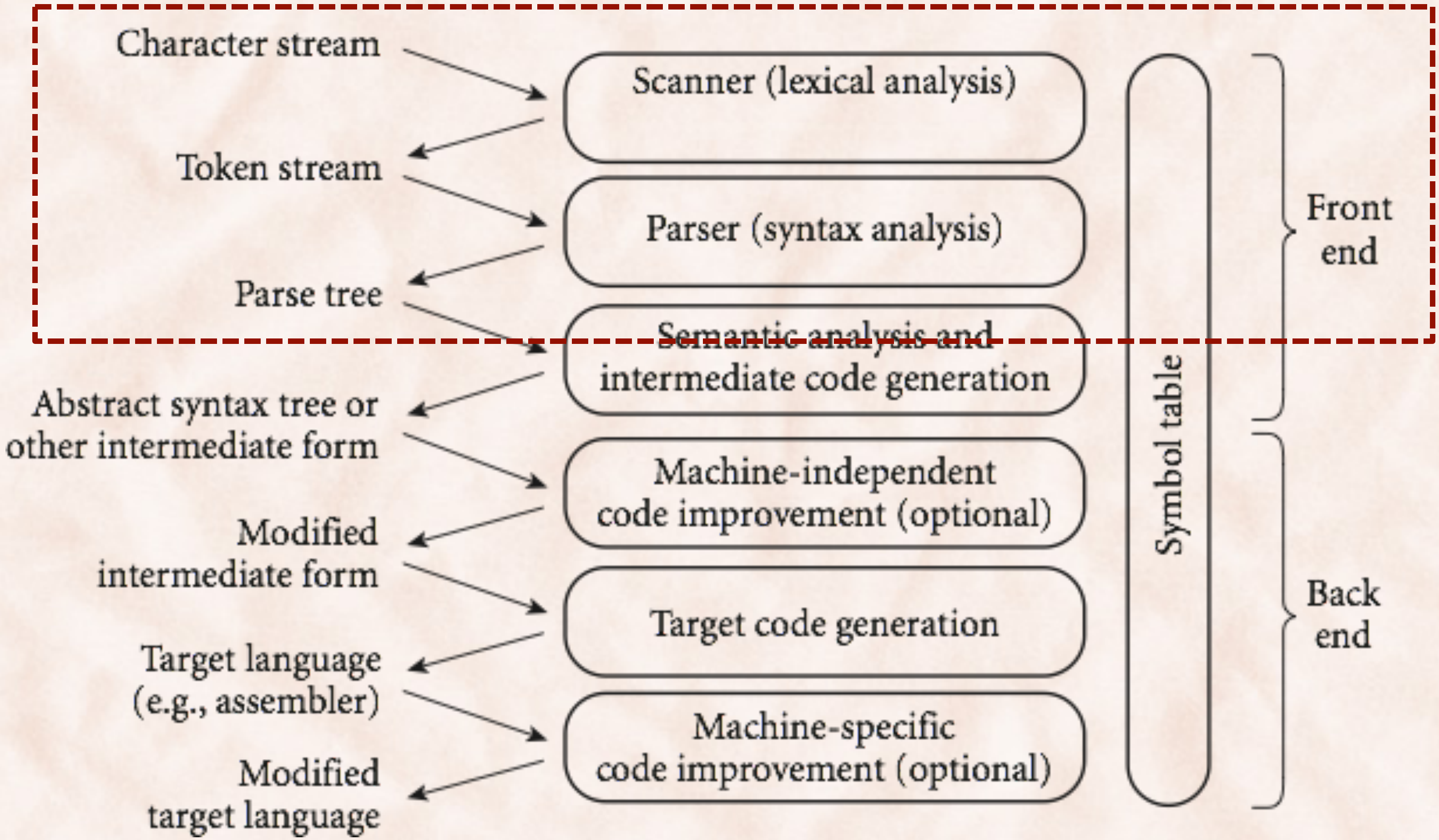
What is a programming language?

- A **programming language** is an artificial language designed for expressing algorithms on a computer:
  - Need to express an *infinite* number of algorithms (Turing complete).
  - Requires an unambiguous **syntax**, specified by a finite *context free grammar*.
  - Should have a well defined compositional **semantics** for each syntactic construct: *axiomatic vs. denotational vs. operational*.
  - Often requires a practical implementation i.e. **pragmatics**:
    - Implementation on a real machine vs. virtual machine
    - *translation vs. compilation vs. interpretation.*

Implementation Methods: Compilation

- Translate high-level program (source language) into machine code (machine language).
- Slow translation, fast execution.
- Compilation process has several phases:
  - **lexical analysis**: converts characters in the source program into lexical units (e.g. identifiers, operators, keywords).
  - **syntactic analysis**: transforms lexical units into *parse trees* which represent the syntactic structure of program.
  - **semantics analysis**: check for errors hard to detect during syntactic analysis; generate *intermediate code*.
  - **code generation**: machine code is generated.

Phases of Compilation



Lexical Analysis: Terminology

- An **alphabet** Σ is a set of characters.
  - the English alphabet.
- A **lexeme** is a string of characters from Σ.
  - index = count + 1;
- A **token** is a category of lexemes:
  - index, count → `identifier`
  - \+ → `plus_operator`
  - 1 → `integer_literal`
  - ; → `semicolon`
- The **lexical rules** of a language specify which lexemes belong to the language, and their categories.

Syntactic Analysis: Terminology

- An **alphabet** Σ is a set of tokens.
  - Σ = {`identifier, plus_operator, integer_literal, …` }
- A **sentence** S is a string of tokens from Σ ($S \in \Sigma^*$).
  - `identifier equal identifier plus_operator integer_literal semicolon`
  - "index = count + 1;" is the original sequence of lexemes.
- A **language** L is a set of sentences ($L \subseteq \Sigma^*$).

- The **syntactic rules** of a language specify which sentences belong to the language:
  - if $S \in L$, then S is said to be *well formed*.

Generative Grammars

- Formal grammars were first studied by linguists:
  - Panini (4th century BC): the earliest known grammar of Sanskrit.
  - Chomsky (1950s): first formalized generative grammars.
- A **grammar** is tuple $G = (\Sigma, N, P, S)$:
  - A finite set Σ of **terminal symbols**.
    - the tokens of a programming language.
  - A finite set N of **nonterminal symbols**, disjoint from Σ.
    - expressions, statements, type declarations in a PL.
  - A finite set P of **production rules**.
    - $P : (\Sigma \cup N)^* \, N \, (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
  - A distinguished **start symbol** $S \in N$.

Generative Grammars

- The language L associated with a formal grammar G is the set of strings from Σ^* that can be generated as follows:
  - start with the start symbol S;
  - apply the production rules in P until no more nonterminal symbols are present.
- Example:
  - $\Sigma = \{a,b,c\}$, $N=\{S,B\}$
  - P consists of the following production rules:
    1. $S \rightarrow aBSc$
    2. $S \rightarrow abc$
    3. $Ba \rightarrow aB$
    4. $Bb \rightarrow bb$

Generative Grammars

- Production rules:
  1. $S \rightarrow aBSc$
  2. $S \rightarrow abc$
  3. $Ba \rightarrow aB$
  4. $Bb \rightarrow bb$
- Derivations of strings in the language L(G):
  - $S \Rightarrow_2 abc$
  - $S \Rightarrow_1 aBSc \Rightarrow_2 aBabcc \Rightarrow_3 aaBbcc \Rightarrow_4 aabbcc$
  - $S \Rightarrow \ldots \Rightarrow aaabbbccc$
- $L(G) = \{a^n b^n c^n \mid n > 0\}$

Chomsky Hierarchy (1956)

- Type 0 grammars (unrestricted grammars)
  - Includes all formal grammars.
- Type 1 grammars (context-sensitive grammars).
  - Rules restricted to: $\alpha A\beta \rightarrow \alpha\gamma\beta$, where A is a non-terminal, and α, β, γ strings of terminals and non-terminals.
- **Type 2** grammars (**context-free grammars**).
  - Rules restricted to $A \rightarrow \gamma$, where A is a non-terminal, and γ a string of terminals and non-terminals
- **Type 3** grammars (**regular grammars**).
  - Rules restricted to $A \rightarrow \gamma$, where A is a non-terminal, and γ:
    - the empty string, or a single terminal symbol followed optionally by a non-terminal symbol.

Context Free Grammars (Type 2)

- Example:
  - $\Sigma = \{a,b\}$, $N=\{S\}$
  - P consists of the following production rules:
    1. $S \rightarrow aSb$
    2. $S \rightarrow \varepsilon$
  - L(G) = ?

CFGs provide the formal **syntax specification** of most programming languages.

Regular Grammars (Type 3)

- Example:
  - $\Sigma = \{a,b\}$, $N=\{S,A,B\}$
  - P consists of the following production rules:
    1. $S \rightarrow aS$
    2. $S \rightarrow cB$
    3. $B \rightarrow bB$
    4. $B \rightarrow \varepsilon$
  - $L(G) = ?$

Regular Grammars/Expressions provide the formal **lexical specification** of most programming languages.

Lexical Analysis

- A lexical analyzer is a "front-end" for the syntactic parser:
  - identifies substrings of the source program that belong together – **lexemes**.
  - lexemes are categorized into lexical categories called **tokens** such as: *keywords, identifiers, operators, numbers, strings, comments*.

- The lexemes of a PL can be formally specified using:
  - Regular Grammars.
  - **Regular Expressions**.
  - Finite State Automata.
  - RE $\Leftrightarrow$ RG $\Leftrightarrow$ FSA (same generative power).

Lexical Analysis: Regular Expressions

lexemes as REs → *tokens*

- Operators:

"+"	{ return (PLUS); } // PLUS = 201
"–"	{ return (MINUS); } // MINUS = 202
"*"	{ return (MULT); } // MULT = 203
"/"	{ return (DIV); } // DIV = 204

- Each keyword is associated a token definition:

"bool"	{ return (BOOL); } // BOOL = 301
"break"	{ return (BREAK); } // BREAK = 302
"case"	{ return (CASE); } // CASE = 303

Lexical Analysis: Regular Expressions

- Identifiers:

  [a–zA–Z_][a–zA–Z_0–9]* { return (ID); } // ID = 200

  - * is Kleene star and means *zero or more.*
  - + means *one or more*
  - . means *any character.*
  - [^\t\n] means any character *other* than whitespaces.

- Numbers:

  [1–9][0–9]* { return (DECIMALINT); }

  0[0–7]* { return (OCTALINT); }

  (0x|0X)[0–9a–fA–F]+ { return (HEXINT); }

Lexical Analysis: Regular Expressions

- More meta-characters:
  - | creates a disjunction of RE's.
    - if A and B are RE's, A|B is an RE that will match either A or B.
  - (…) matches whatever RE is inside the parantheses.
    - indicates the start and end of a *group*
    - | can be used inside groups.

- Regular expressions in Python through module re:
  - http://docs.python.org/3/howto/regex.html
  - http://docs.python.org/3/library/re.html

Lexical Analysis

- In practice, a **scanner generator** (e.g. *Lex, Flex*) reads such lexical definitions and automatically generates code for the lexical analyzer (**scanner**).
- The scanner is implemented as a deterministic **Finite State Automaton (FSA)**.
- An FSA is an abstract state machine that can be used to recognize tokens from a stream of characters.

Syntax: Formal Specification using BNF

- Backus-Naur Form (BNF):
  - Invented by John Backus to describe Algol 60.
  - BNF is a metalanguage notation for Context Free Grammars, used for describing the syntax of programming languages.
    - **Nonterminals** are abstractions for syntactic constructs in the language (e.g. *expressions, statements, type declarations, etc.*)
      - Nonterminals are enclosed in angle brackets.
    - **Terminals** are *lexemes* or *tokens*.

BNF

- Examples:
  ```
  – <if_stmt> → if <logic_expr> then <stmt>
  – <if_stmt> → if <logic_expr> then <stmt> else <stmt>
  ```

  LHS RHS

- The '|' symbol is a logical operator used to specify multiple RHS in a production rule:
  ```
  – <if_stmt> → if <logic_expr> then <stmt>
                 | if <logic_expr> then <stmt> else <stmt>
  ```

Recursive Productions

- Syntactic lists are described using recursion:

```
<ident_list> → ident
             | ident , <ident_list>
```

- Simple expression grammar:

```
<expr> → <expr> + <expr>
       | <expr> * <expr>
       | a | b | c
```

Grammars & Derivations

- A Simple Grammar:

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

- A **derivation** is a repeated application of rules, starting with the start symbol and ending with a **sentence** (a sequence of terminal symbols)

Grammars & Derivations

- An example (leftmost) derivation:

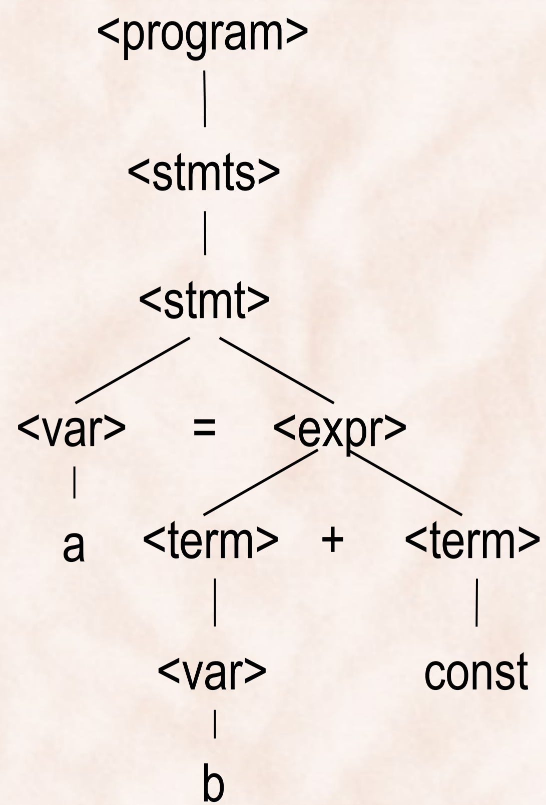
```
<program> => <stmts> => <stmt>
                   => <var> = <expr>
                   => a = <expr>
                   => a = <term> + <term>
                   => a = <var> + <term>
                   => a = b + <term>
                   => a = b + const
```

Derivations

- A string of symbols in a derivation is a *sentential form*.
- A *sentence* is a sentential form that has only terminal symbols.
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded.
- A *rightmost derivation* is one in which the rightmost nonterminal in each sentential form is the one that is expanded.
- A derivation may be neither leftmost nor rightmost

Parse Trees

- **Parse Tree** = a hierarchical representation of a derivation.



Parse Trees

- For any string from L(G), a grammar G defines a recursive tree structure = Parse Tree.

- Parse Trees:
  - The root and intermediate nodes are nonterminals.
  - The leaf nodes are terminals.
  - For each rule used in a derivation step:
    - the LHS is a parent node.
    - the symbols in the RHS are children nodes (from left to right).

Syntactic Ambiguity

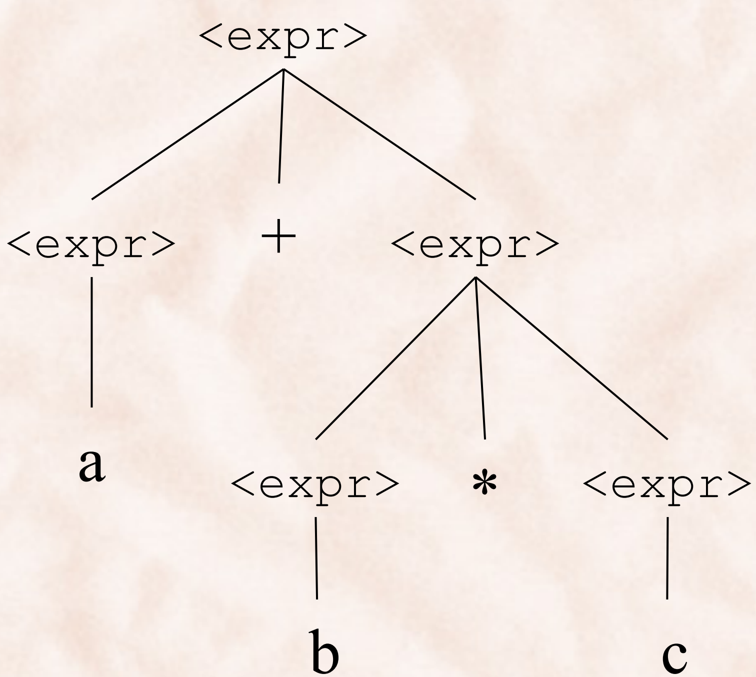
- A grammar is ***ambiguous*** if and only if it can generate a sentence that has two or more distinct parse trees.
- A grammar is ***ambiguous*** if a sentence has more than one leftmost derivations.

- This simple expression grammar is ambiguous :

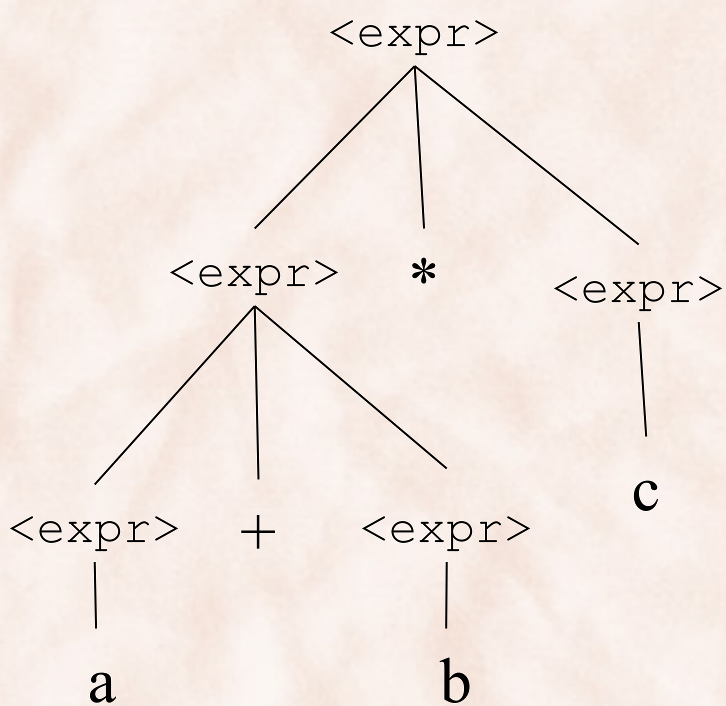
```
<expr> → <expr> + <expr>
       | <expr> * <expr>
       | a | b | c
```

Syntactic Ambiguity

* lower than +



+ lower than *



Operator Precedence

- The expression string “a + b * c” has two different parse trees:
  - Q: Which one is “correct”?
  - A: Both are syntactically correct, but we prefer the first one:
    - Its structure is closer to the the correct semantics of the expression.
    - Want meaning of the expression to be easily determined from its parse tree ⇒ need parse tree to encode precedence rules.
    - Operator ‘*’ generated lower in the parse tree than ‘+’ means that ‘*’ has higher precedence than ‘+’.

Operator Precedence

- Expression grammar that encodes precedence rules:

```
<expr> → <expr> + <term> | <term>
<term> → <term> * <fact> | <fact>
<fact> → a | b | c
```

- What is the parse tree for “a + b * c”?
- What is the parse tree for “a + b + c”?

- Is this new grammar non-ambiguous?

Associativity of operators

- Associativity, like predecedence, can be encoded in the grammar:

```
<expr> → <expr> + <term> | <term>
<term> → <term> * <fact> | <fact>
<fact> → a | b | c
```

  - Left recursive rules ⇒ left associative operators.
  - Right recursive rules ⇒ right associative operators.

- What are the parse trees for “a + b * c” & “a + b + c”?

Associativity of Operators

- Introducing the exponentiation operator ‘^’:

```
<expr> → <expr> + <term> | <term>
<term> → <term> * <fact> | <fact>
<fact> → <base> ^ <fact> | <base>
<base> → a | b | c
```

- What is the precedence of ‘+’, ‘*’, ‘^’?
- What is the associativity of ‘^’?

Syntax vs. Semantics

- Operator precedence and associativity are **semantic rules**.
- CFGs are used to specify **syntactic rules**.

- The grammar can be written to encode semantic rules. Why is this useful?

Syntax vs. Semantics

- The CFG specification is used to build a **Syntactic Analyzer.**
- The Syntactic Analyzer verifies that the input is a *syntactically correct* program.
- The Syntactic Analyzer generates a parse tree that is used in **Intermediate Code Generation** to eventually generate *semantically correct* machine code.
- Hence, the need for parse trees that are both *syntactically correct* and *semantically correct*.

The "Dangling Else" Ambiguity

- Initial grammar rules for the `if-then-else` statement :

```
<if_stmt> → if <logic_expr> then <stmt>
          | if <logic_expr> then <stmt> else <stmt>

<stmt> → <if_stmt>
       | <other_stmt>
```

- Why is this grammar ambiguous?

- Grammar can be rewritten to reflect semantic constraints on the `if-then-else` statement (Example 2.32).

Extended BNF

- Optional parts are placed in brackets [ ]

```
<proc_call> -> ident ['('<expr_list>')']
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

BNF vs. EBNF

- BNF

```
<expr> → <expr> + <term>
             | <expr> - <term>
             | <term>
<term> → <term> * <factor>
             | <term> / <factor>
        | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Syntactic Analysis: The Problem

- **Syntactic Analysis (Parsing)** = a computing problem:
  - Input:
    - a context free grammar.
    - a sequence of tokens.
  - Output:
    - *YES* if the input can be generated by the CFG.
      - The parse tree $\Rightarrow$ need unambiguous grammar.
    - *NO* if the input cannot be generated by the CFG.
      - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly.

Syntactic Analysis: The Algorithms

- **Syntactic Analyzer (Parser) =** an algorithm/program that solves the syntactic analysis problem.
- Time Complexity of syntactic parsing algorithms:
  - Parsers that work for any unambiguous CFG are complex and inefficient – $O(n^3)$:
    - Cocke-Younger-Kasami (CYK) bottom-up parsing algorithm.
  - Compilers use parsers that only work for a subset of all unambiguous CFG grammars, but do it in linear time – $O(n)$:
- Two categories of parsers:
  - Top-down (LL)
  - Bottom-up (LR)

Top-down Parsers

- **Top down** – produce the parse tree, beginning at the root:
  - Traces or builds the parse tree in preorder.
  - Most common are LL(k):
    - L: a left-to-right scanning of the input.
    - L: corresponds to a leftmost derivation.
    - k: number of lookahead symbols.
  - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first k tokens produced by A.
  - Useful parsers look only one token ahead in the input ⇒ LL(1).

Top-down Parsers

- The most common top-down parsing algorithms:
  - **Recursive Descent** – a coded implementation, based directly on the BNF description of the language.
  - Table driven implementation – a parsing table is used to implement the BNF rules.
- Implementation Methods:
  - Manually coded.
  - Generated automatically:
    - ANTLR is an LL(*) parser generator [www.antlr.org].
    - JavaCC is an LL(k) parser generator [javacc.dev.java.net]

Bottom-up Parsing

- **Bottom up** – produce the parse tree, beginning at the leaves:
  - Most common are LR(k):
    - L: a left-to-right scanning of the input.
    - R: corresponds to the reverse of a rightmost derivation.
    - k: number of lookahead symbols.
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation.
  - Useful parsers look only one token ahead in the input $\Rightarrow$ LR(1).
- LR Parser generators:
  - `yacc` (Stephen Johnson for UNIX) ,
  - `bison` (GNU version of yacc).

Recursive Descent Parsing

- Assume we have a lexical analyzer named `lex()`, which puts the next token code in `nextToken`.
- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal.
- The coding process when there is **only one RHS**:
  - For each **terminal symbol** in the RHS, *compare* it with `nextToken`;
    - if they match, continue;
    - else there is an error.
  - For each **nonterminal symbol** in the RHS, *call* its associated parsing subprogram ⇒ problem if grammar is Left Recursive.

Recursive Descent Parsing

- Left Recursive grammar:

```
<expr> → <expr> + <term>
           | <expr> - <term>
           | <term>
<term> → <term> * <factor>
           | <term> / <factor>
           | <factor>
<factor> → id
```

- Cannot do recursive descent parsing:

  `void expr() { expr(); … }` ⇒ infinite recursion!

Recursive Descent Parsing

- An expression grammar that has no left recursion:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id |(<expr>)
```

- Added support for parantheses.

- Left recursion can be eliminated automatically for any CFG.

<expr> → <term> {(+ | -) <term>}

```
void expr() {

/* Parse the first term */

  term();

/* As long as the next token is + or -, call
   lex to get the next token, and parse the
   next term */

  while (nextToken == PLUS ||
         nextToken == MINUS){
    lex();
    term();
  }
}
```

Recursive Descent Parsing

- Convention: Every parsing routine leaves the next token in `nextToken`.
- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse:
  - The correct RHS is chosen on the basis of the next token of input (the lookahead).
    - The next token is compared with the first token that can be generated by each RHS until a match is found.
    - If no match is found, output a syntax error.

<factor> → id|(<expr>)

```
void factor() {
   /* Determine which RHS */
   if (nextToken) == ID)
     /* For the RHS id, just call lex */
     lex();
   else if (nextToken == LEFT_PAREN) {
     lex();
     expr();
     if (nextToken == RIGHT_PAREN)
       lex();
     else
       error();
   }
   else error(); /* Neither RHS matches */
 }
```

The LL Grammars

- The Left Recursion problem:
  - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser.
  - A grammar can be modified to remove direct left recursion.

    For each nonterminal A,

    1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
       where none of the β's begins with A
    2. Replace the original A-rules with:

       $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

       $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$
  - [Aho et al., 1986] give an algorithm to remove left recursion from any CFG.

Eliminating Left Recursion

- Left Recursive grammar:

```
<expr> → <expr> + <term>
           | <expr> - <term>
           | <term>
<term> → <term> * <factor>
           | <term> / <factor>
           | <factor>
<factor> → id
```

- Exercise: Transform into an equivalent grammar w/o left recursion.

The LL Grammars

- The lack of Pairwise Disjointness:
  - The inability to determine the correct RHS on the basis of one token of lookahead
  - Def: FIRST(α) = {a | $\alpha \Rightarrow_* a\beta$ }, where $\Rightarrow_*$ means zero or more derivation steps.
  - [Aho et al., 1986] give an algorithm to compute FIRST(α).
- Pairwise Disjointness Test:
  - For each nonterminal A in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that:

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$$

LL Grammars: Pairwise Disjointness

- Example:

  $A \rightarrow aB \mid bAb \mid Bb$

  $B \rightarrow cB \mid d$

- Example:

  `<variable> → identifier | identifier [<expr>]`

- Pairwise Disjointness hard to solve in general case.
- In some cases, Left Factoring can solve the problem.

LL Grammars: Left Factoring

- Replace:

```
<variable> → identifier |
                  identifier '[' <expr> ']'
```

- With:

```
<variable> → identifier <new>
    <new> → ε | '[' <expr> ']'
```

or

```
<variable> → identifier ['[' <expression> ']']
```

(the outer brackets are metasymbols of EBNF)

Readings & Exercises

- Reading assignment:
  - Chapter 2: Programming Language Syntax:
    - Intro from 2, then 2.1;
    - Intro from 2.2;
    - Intro from 2.3, then 2.3.1, 2.3.2
- Exercises:
  - 2.1, 2.3, 2.9, 2.11, 2.12, 2.13, 2.15 (a-d), 2.27

Summary

- Generative Grammars
  - Regular Grammars (RG) for lexical analysis.
  - Context Free Grammars (CFG) for syntactic analysis.
- Lexical Analysis
  - RG, Regular Expressions.
  - Implementation: Finite State Automata (FSA).
- Syntactic Analysis:
  - CFGs specified using BNF.
  - Implementation:
    - Top-down parsing (e.g. Recursive Descent).
    - Bottom-up Parsing.

Finite State Automata

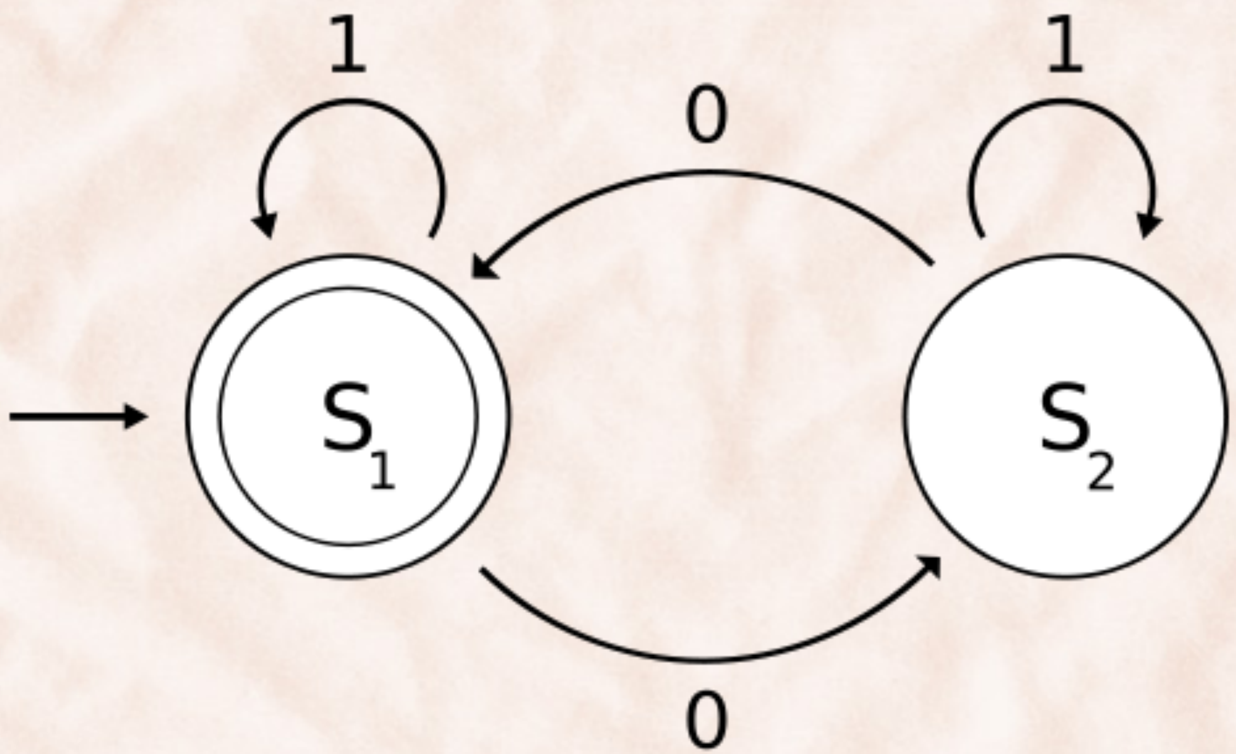
- A deterministic FSA is a tuple $(\Sigma, S, s_0, \delta, F)$:
  - Σ is the input alphabet (a finite set of symbols).
  - S is a finite set of states.
  - $s_0 \in S$ is the initial state.
  - $\delta : S \times \Sigma \rightarrow S$ is the state–transition function.
  - $F \subseteq S$ is the set of final states.

FSA: Representation & Implementation

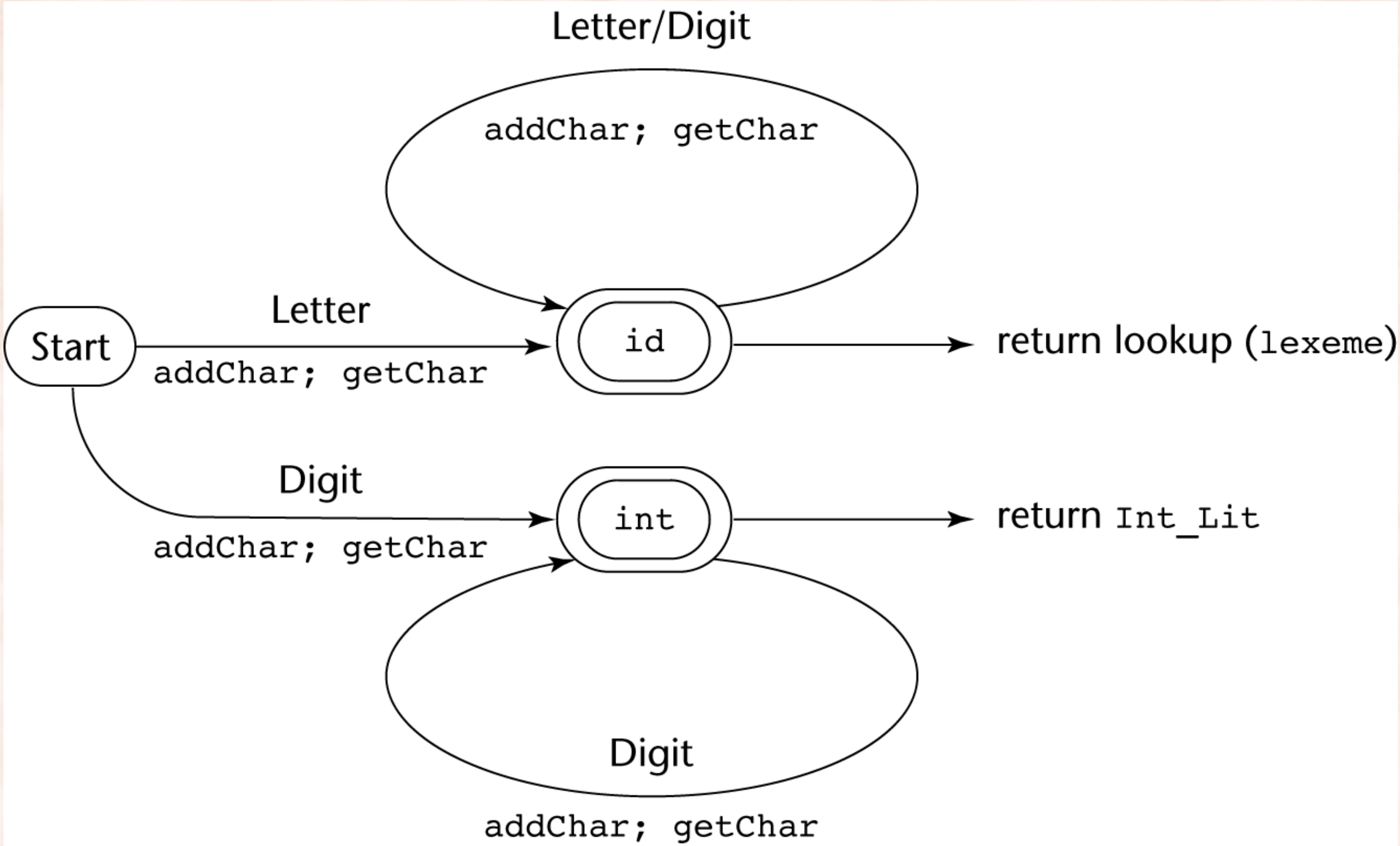
- An FSA can be represented using **transition diagrams**.

- An FSA for recognizing integer literals, identifiers, and reserved words:
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent ⇒ use a character class that includes all letters (`Letter`).
  - When recognizing an integer literal, all digits are equivalent ⇒ use a digit class (`Digit`).
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word.

Transition Diagrams



Transition Diagrams



Syntax vs. Semantics

- **Syntax:** specifies the form or structure of the expressions, statements, and program units.
  - `<if_stmt> → if <logic_expr> then <stmt>`
  - `<if_stmt> → if <logic_expr> then <stmt> else <stmt>`
  - `<while_stmt> → while (<logic_expr>) <stmt>`

- **Semantics:** the meaning of the expressions, statements, and program units.
  - what is the meaning of the Java while statement above?

- Syntax vs. Semantics:
  - semantics should follow directly from syntax.
  - formal specification easier for syntax than for semantics