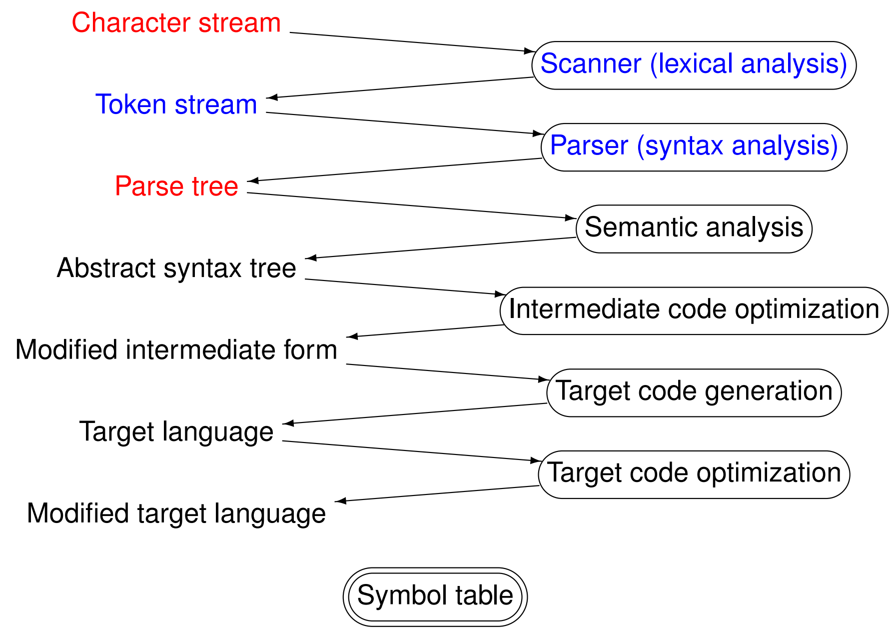
Scanning and Parsing: A Recap

Stefan D. Bruda

CS403, Fall 2024

The Compilation Process



The Lexical Analyzer

- Main role: split the input character stream into tokens
  - Usually even interacts with the symbol table, inserting identifiers in it (especially useful for languages that do not require declarations)
  - This simplifies the design and portability of the parser
- A token is a data structure that contains:
  - The token name = abstract symbol representing a kind of lexical unit
  - A possibly empty set of attributes
- A pattern is a description of the form recognized in the input as a particular token
- A lexeme is a sequence of characters in the source program that matches a particular pattern of a token and so represents an instance of that token
- Most programming languages feature the following tokens
  - One token for each keyword
  - One token for each operator or each class of operators (e.g., relational operators)
  - One token for all identifiers
  - One or more tokens for literals (numerical, string, etc.)
  - One token for each punctuation symbol (parentheses, commas, etc.)

Example of Tokens and Attributes

```
printf("Score = %d\n", score);
```

Lexeme	Token	Attribute
`printf`	**id**	pointer to symbol table entry
`(`	**open_paren**	
`"Score = %d\n"`	**string**	
`,`	**comma**	
`score`	**id**	pointer to symbol table entry
`)`	**cls_paren**	
`;`	**semicolon**	

```
E = M * C ** 2
```

Lexeme	Token	Attribute
`E`	**id**	pointer to symbol table entry
`=`	**assign**	
`M`	**id**	pointer to symbol table entry
`*`	**mul**	
`C`	**id**	pointer to symbol table entry
`**`	**exp**	
`2`	**int_num**	numerical value 2

Specification of Tokens

- Token patterns are simple enough so that they can be specified using regular expressions
- Alphabet Σ: a finite set of symbols (e.g. binary digits, ASCII)
- Strings (not sets!) over an alphabet; empty string: ε
  - Useful operation: concatenation ($\cdot$ or juxtaposition)
  - ε is the identity for concatenation ($\varepsilon w = w\varepsilon = w$)
- Language: a countable set of strings
  - Abuse of notation: For $a \in \Sigma$ we write a instead of $\{a\}$
  - Useful elementary operations: union ($\cup$, $+$, $|$) and concatenation ($\cdot$ or juxtaposition): $L_1 L_2 = L_1 \cdot L_2 = \{w_1 w_2 : w_1 \in L_1 \wedge w_2 \in L_2\}$
  - Exponentiation: $L^n = \{w_1 w_2 \cdots w_n : \forall 1 \leq i \leq n : w_i \in L\}$ (so that $L^0 = \{\varepsilon\}$)
  - Kleene closure: $L^* = \bigcup_{n \geq 0} L^n$
  - Positive closure: $L^+ = \bigcup_{n > 0} L^n$
- An expression containing only symbols from Σ, ε, $\emptyset$, union, concatenation, and Kleene closure is called a regular expression
  - A language described by a regular expression is a regular language

Syntactic Sugar for Regular Expressions

Notation	Regular expression	
r^+	rr^*	one or more instances (positive closure)
$r?$	$r\|\varepsilon$ or $r + \varepsilon$ or $r \cup \varepsilon$	zero or one instance
$[a_1 a_2 \cdots a_n]$	$a_1\|a_2\|\cdots\|a_n$	character class
$[a_1 - a_n]$	$a_1\|a_2\|\cdots\|a_n$	provided that a_1, a_2, $\ldots a_n$ are in sequence
$[\hat{\ }a_1 a_2 \cdots a_n]$ $[\hat{\ }a_1 - a_n]$		anything except $a_1, a_2, \ldots a_n$

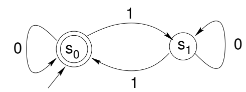
- The tokens in a programming language are usually given as regular definitions = collection of named regular languages

Examples of Regular Definitions

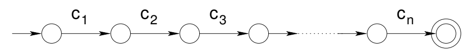
$$
\begin{aligned}
\text{letter\_} &= [A - Za - z\_] \\
\text{digit} &= [0 - 9] \\
\text{id} &= \text{letter\_ (letter\_} \mid \text{digit})^* \\
\text{digits} &= \text{digit}^+ \\
\text{fraction} &= \text{. digits} \\
\text{exp} &= E\ [+-]?\ \text{digits} \\
\text{number} &= \text{digits fraction? exp?} \\
\text{if} &= i\ f \\
\text{then} &= t\ h\ e\ n \\
\text{else} &= e\ l\ s\ e \\
\text{rel\_op} &= <\ \mid\ >\ \mid\ <=\ \mid\ >=\ \mid\ ==\ \mid\ !=
\end{aligned}
$$

State Transition Diagrams

- In order for regular expressions to be used for lexical analysis they must be "compiled" into state transition diagrams
- Also called deterministic finite automata (DFA)
- Finite directed graph
- Edges (transitions) labeled with symbols from an alphabet
- Nodes (states) labeled only for convenience
- One initial state
- Several accepting states (double circles)



- A string $c_1 c_2 c_3 \ldots c_n$ is accepted by a state transition diagram if there exists a path from the starting state to an accepting state such that the sequence of labels along the path is $c_1, c_2, \ldots, c_n$



  - Same state might be visited more than once
  - Intermediate states might be final
- The set of exactly all the strings accepted by a state transition diagram is the language accepted (or recognized) by the state transition diagram

Software Realization

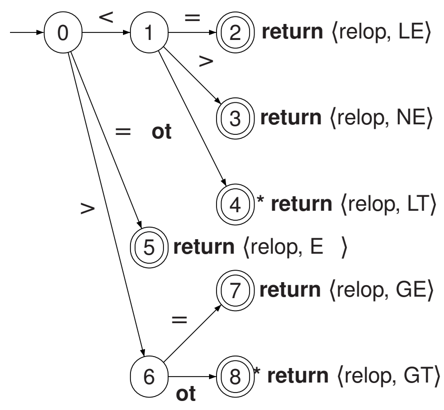
- Big practical advantages of DFA: easy and efficient implementation:
  - Interface to define a vocabulary and a function to obtain the input tokens

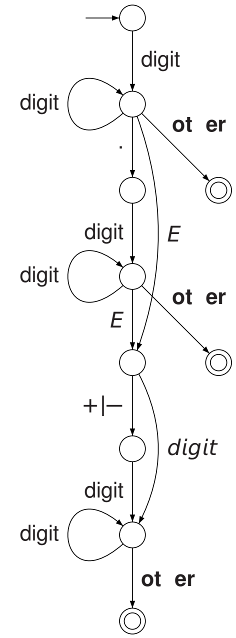
```
typename vocab;       /* alphabet + end-of-string */
const vocab EOS;      /* end-of-string pseudo-token */
vocab getchr(void);   /* returns next symbol */
```

  - Variable (state) changed by a simple `switch` statement as we go along

```
int main (void) {
    typedef enum {S0, S1, ... } state;
    state s = S0;    vocab t = getchr();
    while ( t != EOS ) {
        switch (s) {
            case S0: if (t == ...) s = ...; break;
                     if (t == ...) s = ...; break;
                     ...
            case S1: ...
            ...
        } /* switch */
        t = getchr();   } /* while */
    /* accept iff the current state s is final */
}
```


Examples of State Transition Diagrams





When returning from *-ed states must retract last character

Practical Example: Lex

- The Lex language is a programming language particularly suited for working with regular expressions
  - Actions can also be specified as fragments of C/C++ code
- The Lex compiler compiles the Lex language (e.g., `scanner.l`) into C/C++ code (`lex.yy.c`)
  - The resulting code is then compiled to produce the actual lexical analyzer
  - The use of this lexical analyzer is through repeatedly calling the function `yylex()` which will return a new token at each invocation
  - The attribute value (if any) is placed in the global variable `yylval`
  - Additional global variable: `yytext` (the lexeme)
- Structure of a Lex program:

  Declarations
  %%
  translation rules
  %%
  auxiliary functions

- Declarations include variables, constants, regular definitions
- Transition rules have the form

  Pattern { Action }

  where the pattern is a regular expression and the action is arbitrary C/C++ code

Lex Behaviour

- Lex compile the given regular expressions into one big state transition diagram, which is then repeatedly run on the input
- Lex conflict resolution rules:
  - Always prefer a longer to a shorter lexeme
  - If the longer lexeme matches more than one pattern then prefer the pattern that comes first in the Lex program
- Lex always reads one character ahead, but then retracts the lookahead character upon returning the token
  - Only the lexeme itself in therefore consumed

Context-Free Grammars

- A context-free grammar is a tuple $G = (N, \Sigma, R, S)$, where
  - Σ is an alphabet of terminals
  - N alphabet of symbols called by contrast nonterminals
    - Traditionally nonterminals are capitalized or surrounded by $\langle$ and $\rangle$, everything else being a terminal
  - $S \in N$ is the axiom (or the start symbol)
  - $R \subseteq N \times (N \cup \Sigma)^*$ is the set of (rewriting) rules or productions
    - Common ways of expressing $(\alpha, \beta) \in R$: $\alpha \to \beta$ or $\alpha ::= \beta$
    - Often terminals are quoted (which makes the $\langle$ and $\rangle$ unnecessary)
- Examples:

$$\begin{array}{lcl} \langle\text{exp}\rangle & ::= & CONST \\ & | & VAR \\ & | & \langle\text{exp}\rangle\ \langle\text{op}\rangle\ \langle\text{exp}\rangle \\ & | & (\ \langle\text{exp}\rangle\ ) \\ \langle\text{op}\rangle & ::= & + \mid - \mid * \mid / \end{array}$$

$$\begin{array}{lcl} \langle\text{stmt}\rangle & ::= & ; \\ & | & VAR = \langle\text{exp}\rangle\ ; \\ & | & \texttt{if}\ (\ \langle\text{exp}\rangle\ )\ \langle\text{stmt}\rangle\ \texttt{else}\ \langle\text{stmt}\rangle \\ & | & \texttt{while}\ (\ \langle\text{exp}\rangle\ )\ \langle\text{stmt}\rangle \\ & | & \{\ \langle\text{seq}\rangle\ \} \\ \langle\text{seq}\rangle & ::= & \varepsilon \mid \langle\text{stmt}\rangle\ \langle\text{seq}\rangle \end{array}$$

$$\begin{array}{lcl} \langle\text{balanced}\rangle & ::= & \varepsilon \\ \langle\text{balanced}\rangle & ::= & 0\ \langle\text{balanced}\rangle\ 1 \end{array}$$

Derivations

- $G = (N, \Sigma, R, S)$
- A rewriting rule $A ::= v' \in R$ is used to rewrite its left-hand side (A) into its right-hand side (v'):
  - $u \Rightarrow v$ iff $\exists x, y \in (N \cup \Sigma)^* : \exists A \in N : u = xAy, v = xv'y, A ::= v' \in R$
- Rewriting can be chained ($\Rightarrow^*$, the reflexive and transitive closure of $\Rightarrow$ = derivation)
  - $s \Rightarrow^* s'$ iff $s = s'$, $s \Rightarrow s'$, or there exist strings $s_1, s_2, \ldots, s_n$ such that $s \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow s'$
  - $\langle\text{pal}\rangle \Rightarrow 0\langle\text{pal}\rangle0 \Rightarrow 01\langle\text{pal}\rangle10 \Rightarrow 010\langle\text{pal}\rangle010 \Rightarrow 0101010$

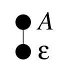
$$\langle\text{pal}\rangle ::= \varepsilon \mid 0 \mid 1 \mid 0\ \langle\text{pal}\rangle\ 0 \mid 1\ \langle\text{pal}\rangle\ 1$$

- The language generated by grammar G: exactly all the terminal strings generated from S: $\mathcal{L}(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$

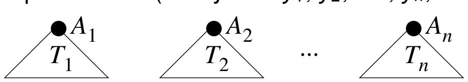
Parse Trees

- Definition:
  1. For every $a \in N \cup \Sigma$ the following is a parse tree (with yield a):

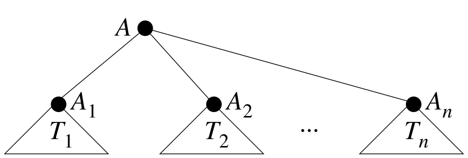
  2. For every $A ::= \varepsilon \in R$ the following is a parse tree (with yield ε):



  3. If the following are parse trees (with yields $y_1, y_2, \ldots, y_n$, respectively):



  and $A ::= A_1 A_2 \ldots A_n \in R$, then the following is a parse tree (w/ yield $y_1 y_2 \ldots y_n$):



- Yield: concatenation of leaves in inorder

Derivations and Parse Trees

- Every derivation starting from some nonterminal has an associated parse tree (rooted at the starting nonterminal)
- Two derivations are similar iff only the order of rule application varies = can obtain one derivation from the other by repeatedly flipping consecutive rule applications
  - Two similar derivations have identical parse trees
  - Can use a “standard” derivation: leftmost ($A \overset{L}{\Rightarrow}{}^* w$) or rightmost ($A \overset{R}{\Rightarrow}{}^* w$)

Theorem

The following statements are equivalent:

- *there exists a parse tree with root A and yield w*
- $A \Rightarrow^* w$
- $A \overset{L}{\Rightarrow}{}^* w$
- $A \overset{R}{\Rightarrow}{}^* w$

- Ambiguity of a grammar: there exists a string that has two derivations that are not similar (i.e., two derivations with diferent parse trees)
  - Can be inherent or not — impossible to determine algorithmically

Inherent Ambiguity in C++ Templates

- Consider the following code:

```
int y;
template <class T> void g(T& v) {
    T::x(y);
}
```

  - The statement `T::x(y)` can be
    - the function call (member function `x` of `T` applied to `y`), or
    - the declaration of `y` as a variable of type `T::x`.
  - Resolution: unless otherwise stated, an identifier is assumed to refer to something that is not a type or template.
    - If we want something else, we use the keyword `typename`:

```
T::x(y);           // function x of T applied to y
typename T::x(y);  // y is a variable of type T::x
```


Parsing

- Interface to lexical analysis:

```
typename vocab;        /* alphabet + end-of-string */
const vocab EOS;       /* end-of-string pseudo-token */
vocab gettoken(void); /* returns next token */
```

- Parsing = determining whether the current input belongs to the given language
  - In practice a parse tree is constructed in the process as well
- General method: Not as efficient as for finite automata
  - Several possible derivations starting from the axiom, must choose the right one
  - Careful housekeeping (dynamic programming) reduces the otherwise exponential complexity to $O(n^3)$
  - We want linear time instead, so we want to determine what to do next based on the next token in the input

Recursive Descent Parsing

- Construct a function for each nonterminal
- Decide which function to call based on the next input token = linear complexity

```
vocab t;

void MustBe (vocab ThisToken) {
    if (t != ThisToken) { printf("reject"); exit(0); }
    t = gettoken();
}
void Balanced (void) {
    switch (t) {
      case EOS:
      case ONE: /* <empty> */
        break;
      default: /* 0 <balanced> 1 */
        MustBe(ZERO);
        Balanced();
        MustBe(ONE);
   }
} /* Balanced */
```

```
int main (void) {
    t = gettoken();
    Balanced();
    /* accept iff
       t == EOS    */
}
```


Recursive Descent Example

```
typedef enum { VAR, EQ, IF, ELSE, WHILE, OPN_BRACE, CLS_BRACE,
               OPN_PAREN, CLS_PAREN, SEMICOLON, EOS            } vocab;
vocab gettoken() {...}
vocab t;
void MustBe(vocab ThisToken) {...}

void Statement();
void Sequence();

int main() {
    t = gettoken();
    Statement();
    if (t != EOS) printf("String not accepted\n");
    return 0;        }
void Sequence() {
    if (t == CLS_BRACE) /* <empty> */ ;
    else { /* <statement> <sequence> */
        Statement();
        Sequence();
    }    }
```

Recursive Descent Example (cont'd)

```
void Statement() {
    switch(t) {
    case SEMICOLON: /* ; */
        t = gettoken();
        break;
    case VAR: /* <var> = <exp> */
        t = gettoken();
        MustBe(EQ);
        Expression();
        MustBe(SEMICOLON);
        break;
    case IF: /* if (<expr>) <statement> else <statement> */
        t = gettoken();
        MustBe(OPEN_PAREN);
        Expression();
        MustBe(CLS_PAREN);
        Statement();
        MustBe(ELSE);
        Statement();
        break;
```

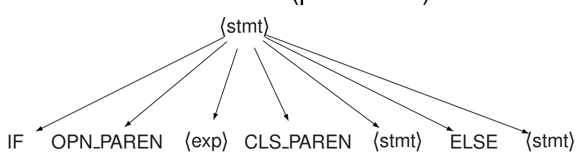

Recursive Descent Example (cont'd)

```
    case WHILE: /* while (exp) <statement> */
        t = gettoken();
        MustBe(OPEN_PAREN);
        Expression();
        MustBe(CLS_PAREN);
        Statement();
        break;
    default: /* { <sequence } */
        MustBe(OPN_BRACE);
        Sequence();
        MustBe(CLS_BRACE);
    } /* switch */
} /* Statement () */
```

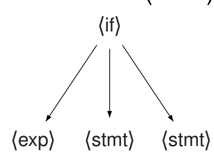

Parse Trees vs. Abstract Syntax Trees

- In practice the output of a parser is a somehow simplified parse tree called abstract syntax tree (AST)
  - Some tokens in the program being parsed have only a syntactic role (to identify the respective language construct and its components)
  - Node information might be augmented to replace them
  - These tokens have no further use and so they are omitted form the AST
  - Other than this omission the AST looks exactly like a parse tree
- Examples of parse trees versus AST

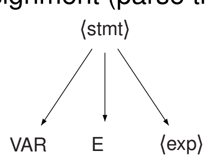
Conditional (parse tree):



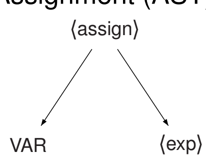
Consitional (AST):



Assignment (parse tree):



Assignment (AST):



Constructing the Parse Tree

- The parse tree/AST can be constructed through the recursive calls:
  - Each function creates a current node
  - The children are populated through recursive calls
  - The current node is then returned

```
class Node {...};

Node* Sequence() {
    Node* current = new Node(SEQ, ...);
    if (t == CLS_BRACE) /* <empty> */ ;
    else { /* <statement> <sequence> */
        current.addChild(Statement());
        current.addChild(Sequence());
    }
    return current;
}
```

Constructing the Parse Tree (cont'd)

```
Node* Statement() {
    Node* current;
    switch(t) {
    case SEMICOLON: /* ; */
        t = gettoken();
        return new Node(EMPTY);
        break;
    case VAR: /* <var> = <exp> */
        current = new Node(ASSIGN, ...);
        current.addChild(VAR, ...);
        t = gettoken();
        MustBe(EQ);
        current.addChild(Expression());
        MustBe(SEMICOLON);
        break;
    case IF: /* if (<expr>) <statement> else <statement> */
        current = new Node(COND, ...);
    /* ... */
    }
    return current;
}
```


Recursive Descent Parsing: Left Factoring

- Not all grammars are suitable for recursive descent:

$$
\begin{array}{rcl}
\langle \text{stmt} \rangle & ::= & \varepsilon \\
& | & \textit{VAR} := \langle \text{exp} \rangle \\
& | & \texttt{IF}\ \langle \text{exp} \rangle\ \texttt{THEN}\ \langle \text{stmt} \rangle\ \texttt{ELSE}\ \langle \text{stmt} \rangle \\
& | & \texttt{WHILE}\ \langle \text{exp} \rangle\ \texttt{DO}\ \langle \text{stmt} \rangle \\
& | & \texttt{BEGIN}\ \langle \text{seq} \rangle\ \texttt{END} \\
\langle \text{seq} \rangle & ::= & \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle\ ;\ \langle \text{seq} \rangle
\end{array}
$$

  - Both rules for $\langle \text{seq} \rangle$ begin with the same nonterminal
  - Impossible to decide which one to apply based only on the next token
  - Fortunately concatenation is distributive over union so we can fix the grammar (left factoring):

$$
\begin{array}{rcl}
\langle \text{seq} \rangle & ::= & \langle \text{stmt} \rangle\ \langle \text{seqTail} \rangle \\
\langle \text{seqTail} \rangle & ::= & \varepsilon \mid\ ;\ \langle \text{seq} \rangle
\end{array}
$$

Recursive Descent Parsing: Ambiguity

- Some programming constructs are inherently ambiguous

$$
\begin{array}{rcl}
\langle \text{stmt} \rangle & ::= & \texttt{if}\ (\ \langle \text{exp} \rangle\ )\ \langle \text{stmt} \rangle \\
& | & \texttt{if}\ (\ \langle \text{exp} \rangle\ )\ \langle \text{stmt} \rangle\ \texttt{else}\ \langle \text{stmt} \rangle
\end{array}
$$

- Solution: choose one path and stick to it (e.g., match the else-statement with the nearest else-less if statement)

```
case IF:
    t = gettoken();
    MustBe(OPEN_PAREN);
    Expression();
    MustBe(CLS_PAREN);
    Statement();
    if (t == ELSE) {
        t = gettoken();
        Statement();
    }
```


Recursive Descent Parsing: Closure, etc.

- Any left recursion in the grammar will cause the parser to go into an infinite loop:

$$\langle \text{exp} \rangle\ ::=\ \langle \text{exp} \rangle\ \langle \text{addop} \rangle\ \langle \text{term} \rangle \mid \langle \text{term} \rangle$$

- Solution: eliminate left recursion using a closure

$$
\begin{array}{rcl}
\langle \text{exp} \rangle & ::= & \langle \text{term} \rangle\ \langle \text{closure} \rangle \\
\langle \text{closure} \rangle & ::= & \varepsilon \\
& | & \langle \text{addop} \rangle\ \langle \text{term} \rangle\ \langle \text{closure} \rangle
\end{array}
$$

  - Not the same language theoretically, but differences not relevant in practice
- This being said, some languages are simply not parseable using recursive descent

$$\langle \text{palindrome} \rangle\ ::=\ \varepsilon \mid 0 \mid 1 \mid 0\ \langle \text{palindrome} \rangle\ 0 \mid 1\ \langle \text{palindrome} \rangle\ 1$$

  - No way to know when to choose the ε rule
  - No way to choose between the second and the fourth rule
  - No way to choose between the third and the fifth rule

Recursive Descent Parsing: Sufficient Conditions

- *first*(α) = set of all initial tokens in the strings derivable from α
- *follow*($\langle N \rangle$) = set of all initial tokens in nonempty strings that may follow $\langle N \rangle$ (possibly including EOS)
- Sufficient conditions for a grammar to allow recursive descent parsing:
  - For $\langle N \rangle ::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ must have $\mathit{first}(\alpha_i) \cap \mathit{first}(\alpha_j) = \emptyset$, $1 \leq i < j \leq n$
  - Whenever $\langle N \rangle \Rightarrow^* \varepsilon$ must have $\mathit{follow}(\langle N \rangle) \cap \mathit{first}(\langle N \rangle) = \emptyset$
- Grammars that do not have these properties may be fixable using left factoring, closure, etc.
- Method for constructing the recursive descent function `N()` for the nonterminal $\langle N \rangle$ with rules $\langle N \rangle ::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$:
  1. For $\alpha_i \neq \varepsilon$ apply the rewriting rule $\langle N \rangle ::= \alpha_i$ whenever the next token in the input is in FIRST(α_i)
  2. For $\alpha_i = \varepsilon$ apply the rewriting rule $\langle N \rangle ::= \alpha_i$ (that is, $\langle N \rangle ::= \varepsilon$) whenever the next token in the input is in FOLLOW($\langle N \rangle$)
  3. Signal a syntax error in all the other cases

Scanning and Parsing

Steps to parse a programming language:

- Construct a scanner
  - Express the lexical structure of the language as regular expressions
  - Convert those regular expressions into a finite automaton (can be automated) = the scanner
- Construct a parser
  - Express the syntax of the language as a context-free grammar
  - Adjust the grammar so that it is suitable for recursive descent
  - Construct the recursive descent parser for the grammar (can be automated) = the parser
- Run the parser on a particular program
  - This implies calls to the scanner to obtain the tokens
  - The result is a parse tree, that will be used in the subsequent steps of the compilation process