



The Phases of a Compiler

- Lexical Analysis – we will cover
- Syntax Analysis – we will cover
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generator



Backus-Naur Form (BNF)

- Also called context free grammar
- Has 4 components
 - A set of tokens known as terminal symbols
 - A set of non-terminals.
 - A set of productions where each production consists of a non-terminal, called the left side of the production, an arrow, and a sequence of tokens and/or non-terminals, called the right side of the production
 - A designation of one of the non-terminals as the start symbol.



Tokens and non-terminals

- Are keywords, semicolons, and other lexical elements
- non-terminals are variables that may represent a sequence of tokens
- The start symbol is normally just the first line in the grammar



A Grammar

- $9+5-2$
- The following grammar describes the syntax of the above expression:

```
list -> list + digit
list -> list - digit
list -> digit
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



What are?

- The tokens?
- The nonterminals?
- The start symbol?



Once we have a Grammar

- We can create a parse tree from it!
- Parse trees pictorially show how the start symbol of a grammar derives a string in the language.
- Finding a parse tree for a given string of tokens is called “parsing” that string.



Context Free Grammar

- In a context free grammar, a parse tree is a tree with the following properties:
 - The root is labeled by the start symbol.
 - Each leaf is labeled by a token or E (empty)
 - Each interior node is labeled by a nonterminal.
 - If A is the nonterminal labeling some interior node and $X_1, X_2, X_3, \dots, X_n$ are the labels of the children of that node from left to right, then $A \rightarrow X_1 X_2 X_3 \dots X_n$ is a production.



Ambiguity

- Some grammars have more than 1 parse tree for a given string
- Example:

$string \rightarrow string + string \mid string - string$
 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Associativity

- By convention: $9+5 + 2$ is equivalent to $(9+5) + 2$
- This is left associative. Left associative trees grow to the left and right associative trees grow to the right.
- Some operators are right associative
 - In C, $a=b=c$ is treated like $a=(b=c)$
 - $a=b=c$ may be generated by the following grammar:


```
right -> letter = right | letter
letter -> a | b | ... | z
```



Precedence

- $9+5*2$ can be evaluated as $(9+5)*2$ or as $9+(5*2)$. Associativity doesn't resolve this ambiguity!
- * has a higher precedence if * takes its operands before +
- We can create 2 nonterminals *expr* and *term* for the two levels of precedence and an extra nonterminal factor for generating basic units in expressions.



The Grammar

```
expr -> expr + term
      | expr - term
      | term
factor -> digit | ( expr )
term -> term * factor
      | term / factor
      | factor
```



For the language statements shown...

- What are the terminals used?
- Write the parse tree for the terms
- Come up with a grammar that explains the following statements and allows for
 - multiple digits in a row
 - Precedence where appropriate

```
(set x (* 1 2))
(set x (+ x 5))
```

```
int x = 1*2;
x = x + 5;
```

```
$x = 1 * 2;
$x = $x + 5;
```



The Java Grammar

Java snippet:

```
int x = 1 * 2;
x = x + 5;
```

Terminals used:

```
id
int
0 1 2 3 4 5 6 7 8 9
+ - * / ( ) ; =
```

Grammar:

```
statement -> declaration assignment
declaration -> primitiveType id ;
                | primitiveType id = expr ;
assignment -> id = expr ;
primitiveType -> int
expr -> expr + term | expr - term | term
term -> term * factor | term / factor | factor
factor -> digits | id | ( expr )
digits -> digit | digits digit
digit -> 0|1| ...
```



Perl Grammar

Perl snippet:

```
$x = 1 * 2;
$x = $x + 5;
```

Terminals used:

```
id
0 1 2 3 4 5 6 7 8 9
+ - * / ( ) ; = $
```

Grammar:

```
assignments -> assignments assignment /
                assignment
assignment -> scalarId = expr ;
scalarId -> $ id
expr -> expr + term | expr - term | term
term -> term * factor | term / factor | factor
factor -> digits | scalarId | ( expr )
digits -> digit | digits digit
digit -> 0|1| ...
```



Lisp Grammar

Lisp snippet:

```
(setf x (* 1 2))
(setf x (+ x 5))
```

Terminals used:

```
symbol
setf
0 1 2 3 4 5 6 7 8 9
+ - * / ( )
```

Grammar:

```
functions -> functions function | function
function -> ( operator operand operand )
operator -> setf | * | - | / | +
operand -> symbol | digits | function
digits -> digit | digits digit
digit -> 0|1| ...
```

Can you find a problem with this?



The Real Grammar Takes More Into Account

From the Java language specs:

Identifier: IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

IdentifierChars: JavaLetter
IdentifierChars JavaLetterOrDigit

JavaLetter: any Unicode character that is a Java letter (see below)

JavaLetterOrDigit: any Unicode character that is a Java letter-or-digit (see below)



More Java Comment Defs

- Comment: TraditionalComment | EndOfLineComment | DocumentationComment
- TraditionalComment: /* NotStar CommentTail
- EndOfLineComment: // CharactersInLineopt LineTerminator
- DocumentationComment: /* * CommentTailStar
- CommentTail: * CommentTailStar | NotStar CommentTail
- CommentTailStar: /* * CommentTailStar | NotStarNotSlash | CommentTail
- NotStar: InputCharacter but not * | LineTerminator
- NotStarNotSlash: InputCharacter but not * or / | LineTerminator
- CharactersInLine: InputCharacter | CharactersInLine InputCharacter