Chapter 2
A Simple One – Pass Compiler

The Entire Compilation Process

- Grammars for Syntax Definition
- Syntax-Directed Translation
- Parsing - Top Down & Predictive
- Pulling Together the Pieces
- The Lexical Analysis Process
- Symbol Table Considerations
- A Brief Look at Code Generation
- Concluding Remarks/Looking Ahead

Overview

Programming Language can be defined by describing

1. The syntax of the language
    1. ***What its program looks like***
    2. ***We use CFG or BNF (Backus Naur Form)***
2. The semantics of the language
    1. ***What its program mean***
    2. ***Difficult to describe***
    3. ***Use informal descriptions and suggestive examples***

Grammars for Syntax Definition

- A Context-free Grammar (CFG) Is Utilized to Describe the Syntactic Structure of a Language
- A CFG Is Characterized By:

1. A Set of Tokens or Terminal Symbols
2. A Set of Non-terminals
3. A Set of Production Rules
   Each Rule Has the Form

   $$NT \rightarrow \{T, NT\}^*$$

4. A Non-terminal Designated As the Start Symbol

Grammars for Syntax Definition Example CFG

$list \rightarrow list + digit$

$list \rightarrow list - digit$

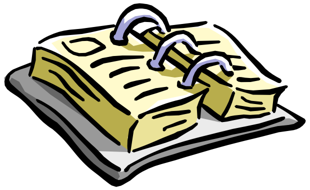
$list \rightarrow digit$

$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

(the "|" means OR)

(So we could have written

$list \rightarrow list + digit \mid list - digit \mid digit$ **)**



Information

- A string of tokens is a sequence of zero or more tokens.
- The string containing with zero tokens, written as $\in$, is called ***empty*** string.
- A grammar derives ***strings*** by beginning with the start symbol and repeatedly replacing the non terminal by the right side of a production for that non terminal.
- The token strings that can be derived from the start symbol form the ***language*** defined by the grammar.

Grammars are Used to Derive Strings:

Using the CFG defined on the earlier slide, we can derive the string: 9 - 5 + 2 as follows:

list → *list* + *digit*	P1 : *list* → *list* + *digit*
→ *list* - *digit* + *digit*	P2 : *list* → *list* - *digit*
→ *digit* - *digit* + *digit*	P3 : *list* → *digit*
→ 9 - *digit* + *digit*	P4 : *digit* → 9
→ 9 - 5 + *digit*	P4 : *digit* → 5
→ 9 - 5 + 2	P4 : *digit* → 2

Grammars are Used to Derive Strings:

This derivation could also be represented via a Parse Tree (parents on left, children on right)

$list \rightarrow list + digit$

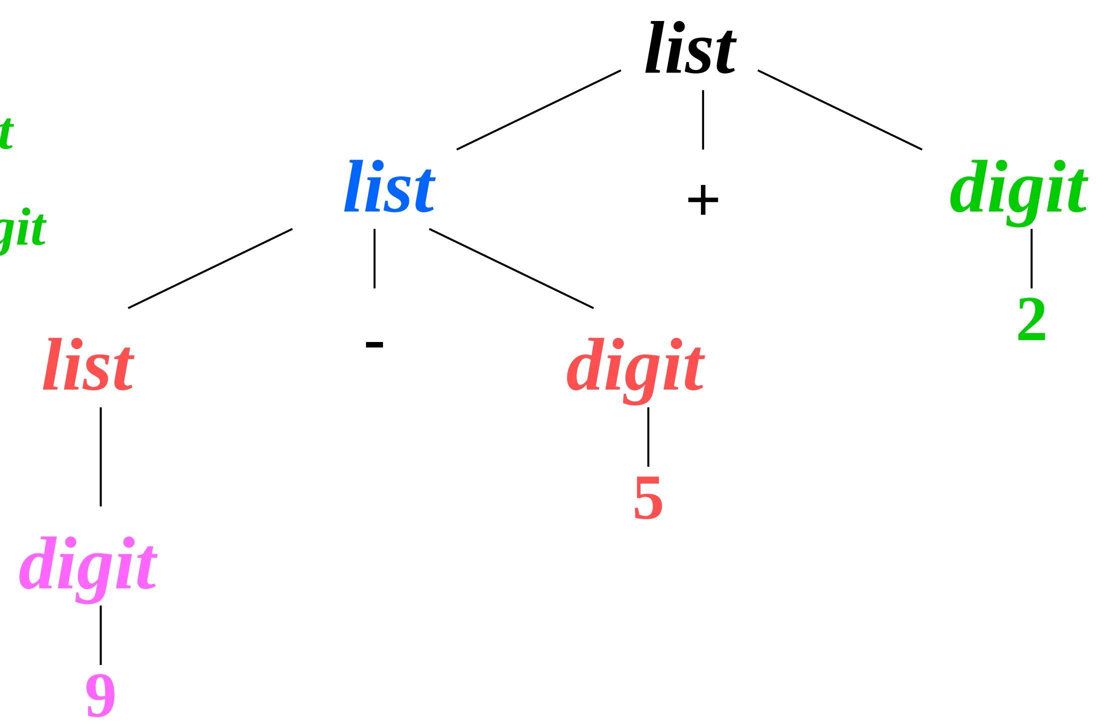
$\rightarrow list - digit + digit$

$\rightarrow digit - digit + digit$

$\rightarrow 9 - digit + digit$

$\rightarrow 9 - 5 + digit$

$\rightarrow 9 - 5 + 2$



Defining a Parse Tree

- A parse tree pictorially shows **how** the start symbol of a grammar derives a string in the language.
- More Formally, a Parse Tree for a CFG Has the Following Properties:
  - Root Is Labeled With the Start Symbol
  - Leaf Node Is a Token or $\in$
  - Interior Node Is a Non-Terminal
  - If $A \rightarrow x1x2\ldots xn$, Then A Is an Interior; $x1x2\ldots xn$ Are Children of A and May Be Non-Terminals or Tokens

Other Important Concepts
Ambiguity

Two derivations (Parse Trees) for the same token string.

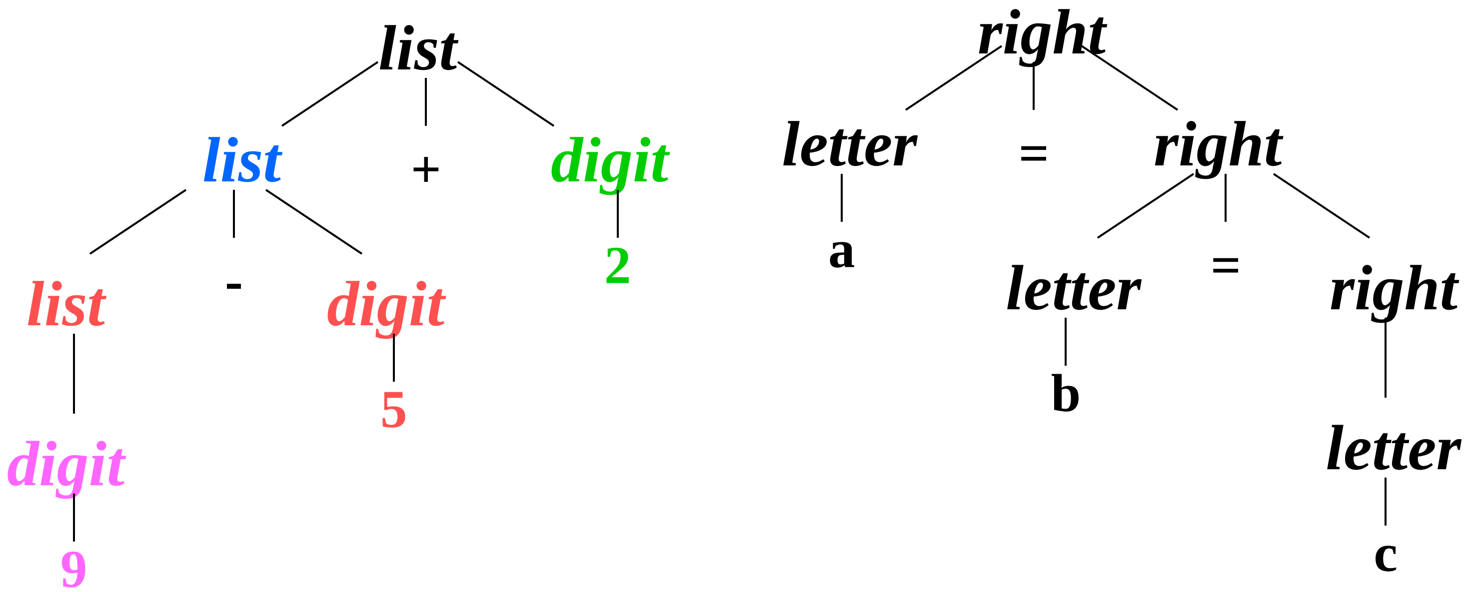
Grammar:

string → *string* + *string* | *string* – *string* | 0 | 1 | …| 9

Why is this a Problem ?

Other Important Concepts
Associativity of Operators

Left vs. Right



list → *list* + *digit* |

| *list* - *digit* | *digit*

digit → 0 | 1 | 2 | …| 9

right → *letter* = *right* | *letter*

letter → a | b | c | …| z

Embedding Associativity

- The language of arithmetic expressions with + -
  - (ambiguous) grammar that does not enforce associativity

    $string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid \ldots \mid 9$

  - non-ambiguous grammar enforcing left associativity (parse tree will grow to the left)

    $string \rightarrow string + digit \mid string - digit \mid digit$

    $digit \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$

  - non-ambiguous grammar enforcing right associativity (parse tree will grow to the right)

    $string \rightarrow digit + string \mid digit - string \mid digit$

    $digit \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$

Other Important Concepts
Operator Precedence

What does

9 + 5 * 2

mean?

Typically

( )

* /

\+ -

is precedence order

This can be incorporated into a grammar via rules:

$$
\begin{aligned}
\textit{expr} &\rightarrow \textit{expr} + \textit{term} \mid \textit{expr} - \textit{term} \mid \textit{term} \\
\textit{term} &\rightarrow \textit{term} * \textit{factor} \mid \textit{term} / \textit{factor} \mid \textit{factor} \\
\textit{factor} &\rightarrow \text{digit} \mid (\ \textit{expr} \ ) \\
\text{digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9
\end{aligned}
$$

Precedence Achieved by:

expr & *term* for each precedence level

Rules for each are left recursive <u>or</u> associate to the left

Syntax for Statements

$$
\begin{aligned}
\textit{stmt} \rightarrow\ & \text{id} := \textit{expr} \\
& \mid \text{if } \textit{expr} \text{ then } \textit{stmt} \\
& \mid \text{if } \textit{expr} \text{ then } \textit{stmt} \text{ else } \textit{stmt} \\
& \mid \text{while } \textit{expr} \text{ do stmt} \\
& \mid \text{begin } \textit{opt\_stmts} \text{ end}
\end{aligned}
$$

Ambiguous Grammar?

Syntax-Directed Translation

- Associate Attributes With Grammar Rules and Translate as Parsing occurs
- The translation will follow the parse tree structure (and as a result the structure and form of the parse tree will affect the translation).
- First example: Inductive Translation.
- Infix to Postfix Notation Translation for Expressions
  - Translation defined inductively as: Postfix(E) where E is an Expression.

Rules

1. ***If*** **E** ***is a*** **variable** ***or*** **constant** ***then*** **Postfix(E) = E**
2. ***If*** **E** ***is*** **E1 op E2** ***then*** **Postfix(E)**

   = Postfix(E1 op E2) = Postfix(E1) Postfix(E2) op
3. ***If*** **E** ***is*** **(E1)** ***then*** **Postfix(E) = Postfix(E1)**

Examples

Postfix((9 – 5) + 2)
= Postfix((9 – 5)) Postfix(2) +
= Postfix(9 – 5) Postfix(2) +
= Postfix(9) Postfix(5) - Postfix(2) +
= 9 5 – 2 +

Postfix(9 – (5 + 2))
= Postfix(9) Postfix((5 + 2)) -
= Postfix(9) Postfix(5 + 2) –
= Postfix(9) Postfix(5) Postfix(2) + –
= 9 5 2 + –

Syntax-Directed Definition

- Each Production Has a Set of Semantic Rules
- Each Grammar Symbol Has a Set of Attributes
- For the Following Example, String Attribute "*t*" is Associated With Each Grammar Symbol

$$\boxed{\begin{array}{l} \textbf{\textit{expr}} \rightarrow \textbf{\textit{expr}} - \textbf{\textit{term}} \mid \textbf{\textit{expr}} + \textbf{\textit{term}} \mid \textbf{\textit{term}} \\ \textbf{\textit{term}} \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \ldots \mid \mathbf{9} \end{array}}$$

- recall: What is a Derivation for 9 + 5 - 2?

list → ***list* - *digit*** → ***list* + *digit* - *digit*** → ***digit* + *digit* - *digit***
→ **9 + *digit* - *digit*** → **9 + 5 - *digit*** → **9 + 5 - 2**

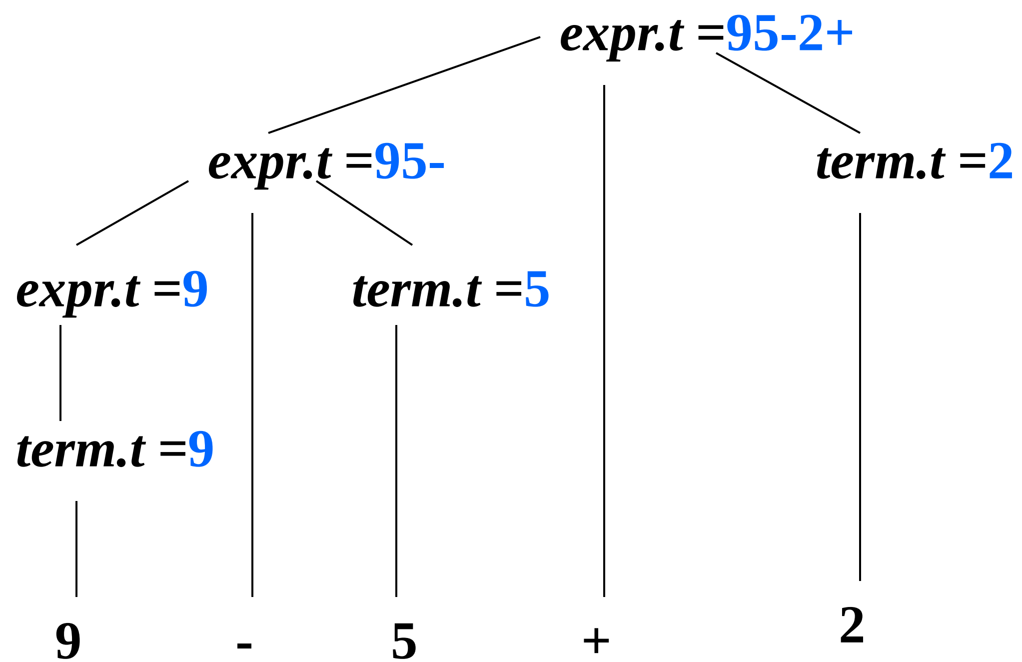
Syntax-Directed Definition (2)

- Each Production Rule of the CFG Has a Semantic Rule

Production	Semantic Rule
expr → *expr* + *term*	*expr.t* := *expr.t* || *term.t* || '+'
expr → *expr* – *term*	*expr.t* := *expr.t* || *term.t* || '-'
expr → *term*	*expr.t* := *term.t*
term → 0	*term.t* := '0'
term → 1	*term.t* := '1'
….	….
term → 9	*term.t* := '9'

- Note: Semantic Rules for *expr* define *t* as a "synthesized attribute" i.e., the various copies of *t* obtain their values from "children *t*'s"

Semantic Rules are Embedded in Parse Tree



- It starts at the root and recursively visits the children of each node in left-to-right order
- The semantic rules at a given node are evaluated once all descendants of that node have been visited.
- A parse tree showing all the attribute values at each node is called annotated parse tree.

Translation Schemes

Embedded Semantic Actions into the right sides of the productions.

A translation scheme is like a syntax-directed definition except the order of evaluation of the semantic rules is explicitly shown.

***expr* → *expr* + *term* {*print*('+')}**

→ *expr* - *term* {*print*('-')}

→ *term*

***term* → *0* {*print*('0')}**

***term* → 1 {*print*('1')}**

…

***term* → 9 {*print*('9')}**