Syntax directed translation

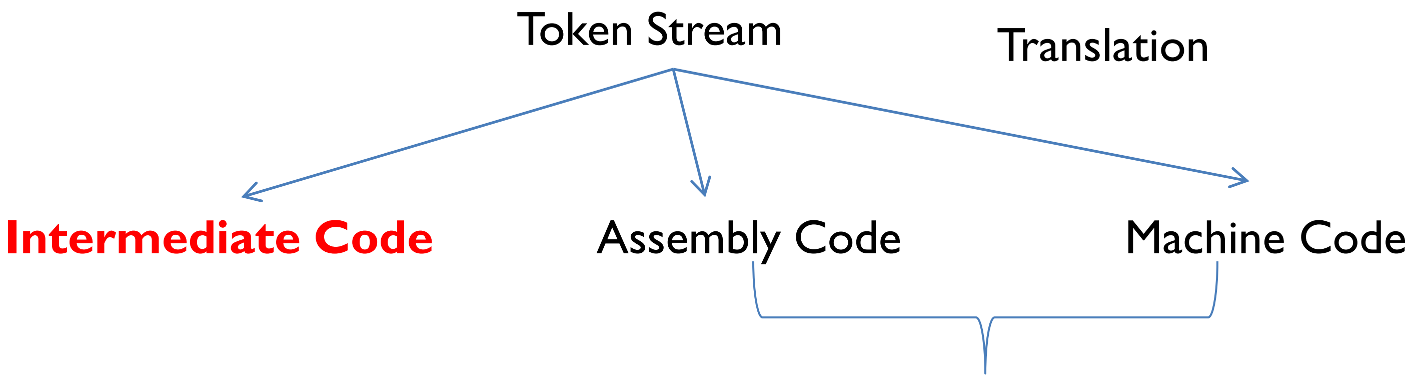
Introduction

- Regular expression and Context Free Grammar provides a useful notation for creating programs for the initial phases of compilation.
- There is, however a notational framework for intermediate code generation that is an extension of context free grammars
- This framework is called *syntax-directed translation scheme*
  - allows **subroutines** or “semantic actions” to be attached to the *productions* of a context free grammar

- The syntax directed translation scheme is useful because it enables us to express the generation of intermediate code directly in terms of the syntactic structure of the source language

Technique

- Translation of basic programming language constructs like Arithmetic expressions, Boolean expressions, Case structure.



Token Stream

Translation

Intermediate Code

Assembly Code

Machine Code

Positives :

Speeds up compilation

Negatives :

Difficult to write

Optimization is harder

Technique

1. **Translation from token streams to intermediate code**
2. Translation to machine or assembly code

- One form of intermediate code called three address code will be used here
- Implementation using quadruples

Syntax –Directed Translation Schemes

- In designing an intermediate code generator there are two basic issues
  - Determine intermediate code for each programming construct.
  - Implement algorithm for generating this code.

Semantic actions

- We shall use the formalism of syntax directed translation schemes to describe the output we wish to generate for each input construct
- a syntax directed translation scheme is
  - a context-free grammar in which a program fragment called an output action/semantic action/semantic rule is associated with each production

- Example:

  Suppose output action α is associated with production A → XYZ.

- Bottom-up parsing
  - when the parser recognizes in its input a substring *w* which has a derivation $A \Rightarrow XYZ \Rightarrow^* w$
  - This means, α is executed whenever XYZ is reduced to A
- Top-down parsing

  This means, α is executed whenever A, X, Y or Z is expanded whichever is appropriate

- The output action may involve
  - computation of values for variables belonging to the compiler
  - the generation of intermediate code
  - the printing of an error diagnostic
  - the placement of some value in a table

Translation

- A *value* associated with a grammar symbol is called a ***translation*** of that symbol
  - The translation may be a **structure** consisting of fields of various types
  - The rules for computing these values/translations can be as involved as we wish
  - we denote the translation *fields* of a grammar symbol *X* with names such as X.VAL, X.TRUE and so forth

- If we have a production with several instances of the same symbol on the right, superscripts can be used to distinguish them
  - $E \rightarrow E^{(1)} + E^{(2)}$ $\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$

    Attribute of E with name VAL

    Semantic Action
  - the semantic action is enclosed in braces and appears after the production
- The terminal symbol + is translated to its usual meaning by the semantic rule.
- In most compilers, we need an action to generate code to perform the addition
- This translation is suitable for a “desk calculator”

Synthesized translation

- If the value of a translation of the non-terminal on the left-hand side of a production is a function of the translations of the non-terminals in the right-hand side, such a translation is called **synthesized translation**.
- Eg. E → E(1) + E(2) {E.VAL := E(1).VAL + E(2).VAL }

Inherited translation

- When the translation of a nonterminal on the right side of the production is defined in terms of a translation of the nonterminal on the left is called **inherited translation**
- **A→XYZ {Y.VAL := 2 * A.VAL }**
- Synthesized translation are more natural than inherited translation for mapping programming languages constructs to intermediate code.

Translation on the parse tree

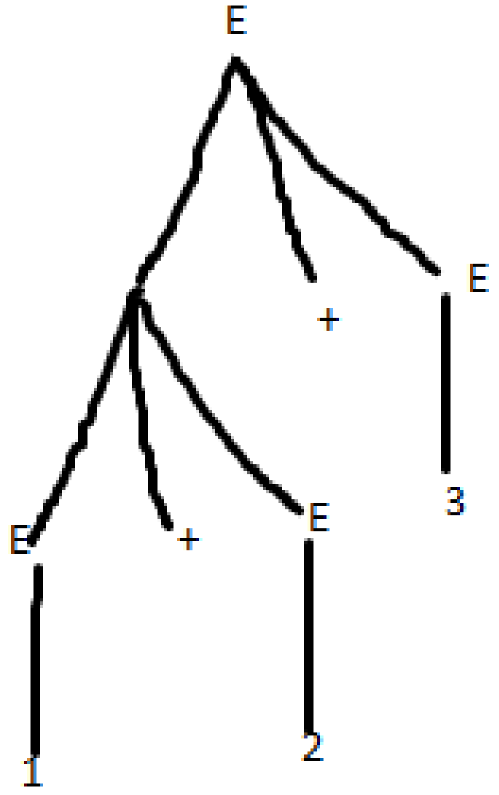
- parse tree for expression 1+2+3
- Grammar

  Production semantic action

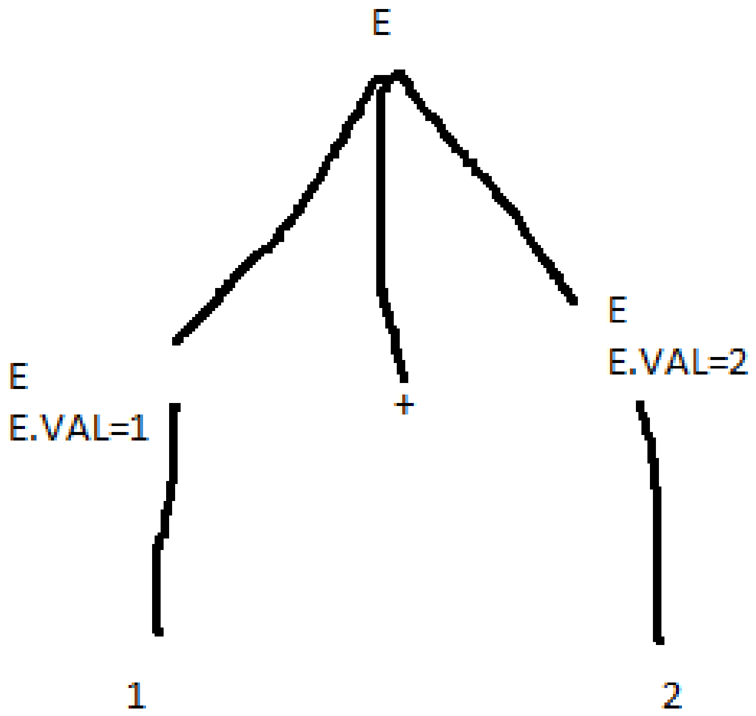
  - $E \rightarrow E^{(1)} + E^{(2)}$ { $E.VAL:=E^{(1)}.VAL+E^{(2)}.VAL$}
  - $E \rightarrow$ ***digit*** *{E.VAL:=****digit****}*

- when the formulas are defined , we must make sure that these formulas will work for all possible legal combinations of productions.
- If we a have a grammar symbol on the left it should work when it is used on the right side of the production.
- Their values are placed on the stack.

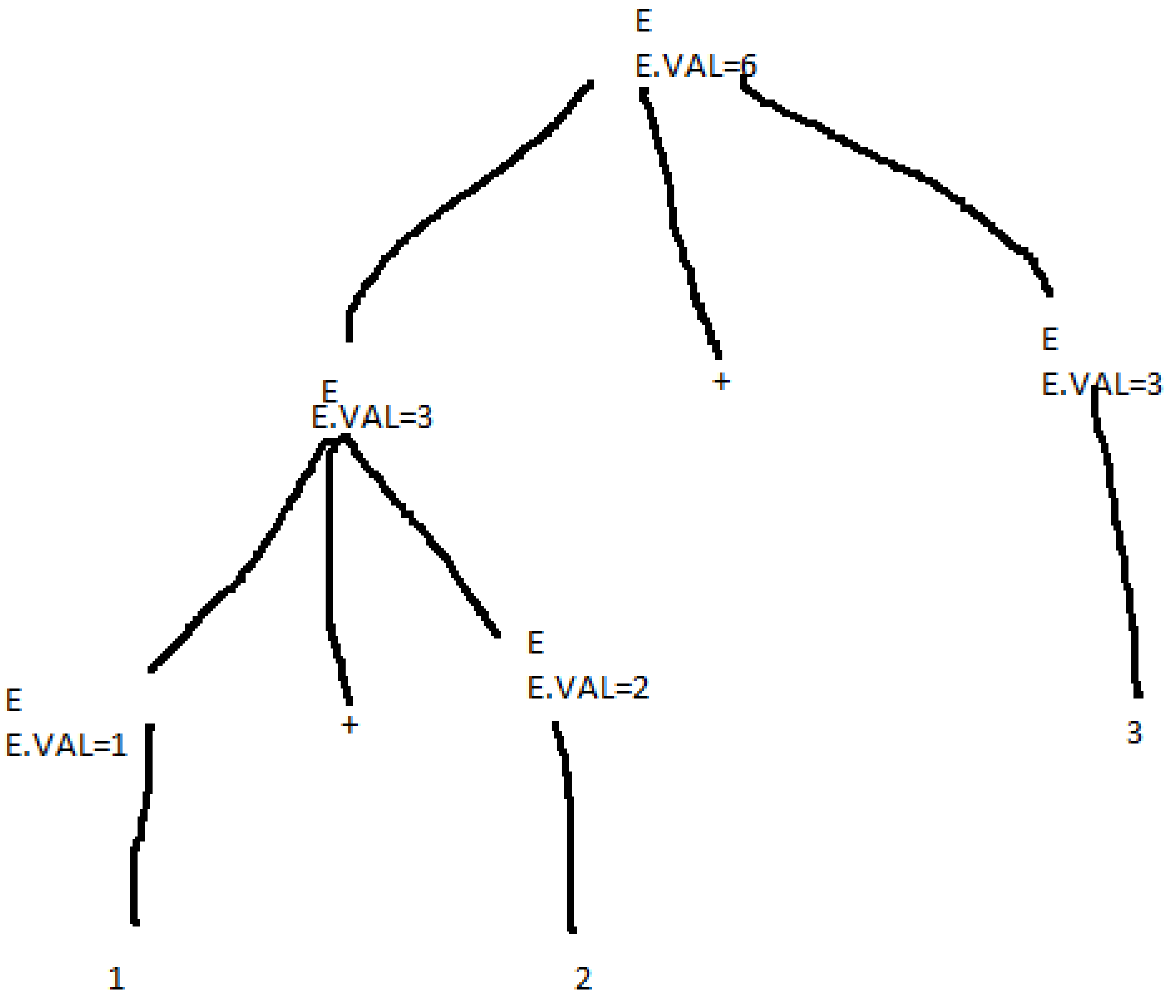
Parse tree for expression 1+2+3



- Subtree with computed translation



Complete parse tree


E
E.VAL=6
E
E.VAL=3
+
E
E.VAL=3
E
E.VAL=1
+
E
E.VAL=2
3
1
2