Crafting a Compiler with C (VII)

資科系
林偉川

CGF Definition

Grammar and parsing !!

CFG G=(V,T,S,P) is 4-tuple, where

V (V_n) : a finite set of variables(Non-terminal)

T (V_t) : a finite set of terminal symbols (token set produced by the scanner)

$S \in V_n$ is a special symbol called ***start*** symbol

P is a finite set of productions (rules) where

$A \rightarrow X_1 \ldots X_m$, where $A \in V_n$, $X_i \in V_n \cup V_t$

2

CFG notation

- a,b,c in lowercase denote symbols in V_t
- A,B,C in uppercase denote symbols in V_n
- α,β,γ denote strings in V*
- u,v,w denote strings in V_t*
- $A \rightarrow \alpha$ or $A \rightarrow X_1 \ldots X_m$, means the left-hand side of a production should be a single nonterminal, but the right-hand side is a string of zeor or more vocabulary symbols
- $A \rightarrow \alpha \mid \beta \mid \ldots \mid \zeta$ is an EBNF

3

CFG derivation

- $A \rightarrow \gamma$ is a production, then $\alpha A \beta \Rightarrow \alpha \gamma \beta$ denote a one-step derivation, extend $\rightarrow$ to $\Rightarrow^+$, derived in one or more steps, and $\Rightarrow^*$, derived in zero or more steps
- $S \Rightarrow^* \beta$, then β is said to be a sentential form of the CFG. SF(G) is the set of sentential forms of grammar G, $L(G)=\{x \in V_t^* \mid S \Rightarrow^+ x\}$, is the produced language $\Rightarrow$ $L(G)=SF(G) \cap V_t^*$ the language G is simply those sentential forms of G that are terminal strings

4

CFG derivation

- If $G_1 \neq G_2$, and $L(G_1)=L(G_2)$, then G_1 and G_2 are equivalence
- When deriving a token sequence, if more than one non-terminal is present, we have a choice of which non-terminal to expand next and what production is applied
- Leftmost derivation denoted as $\rightarrow_{lm}$, $\rightarrow_{lm}+$ and $\rightarrow_{lm}*$ (top-down parser)
- Rightmost derivation denoted as $\rightarrow_{rm}$, $\rightarrow_{rm}+$ and $\rightarrow_{rm}*$ (bottom-up parser)

5

Derivation example

A grammar G as left part (V is variables, F is functions), the leftmost and rightmost derivation of F(V+V) is denoted as middle and right part.

E→Prefix (E)
E→V Tail
Prefix→F
Prefix →λ
Tail →+ E
Tail→λ

$E \rightarrow_{lm}$Prefix(E)
$E \rightarrow_{lm}$ F(E)
$E \rightarrow_{lm}$ F(V Tail)
$E \rightarrow_{lm}$ F(V+E)
$E \rightarrow_{lm}$ F(V+V Tail)
$E \rightarrow_{lm}$ F(V+V)

$E \rightarrow_{rm}$Prefix(E)
$E \rightarrow_{rm}$ Prefix(V Tail)
$E \rightarrow_{rm}$ Prefix(V+E)
$E \rightarrow_{rm}$ Prefix(V+V Tail)
$E \rightarrow_{rm}$ Prefix(V+V)
$E \rightarrow_{rm}$ F(V+V)

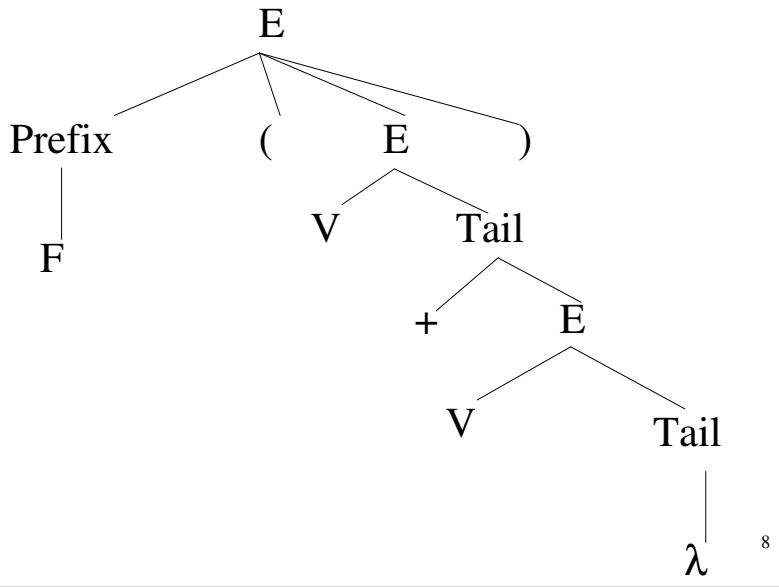
6

Leftmost or rightmost derivation

- A sentential form produced via a leftmost derivation sequence is called left sentential form
- An alternative to a leftmost derivation is a rightmost derivation (canonical derivation)
- A bottom-up parser discovers the production used to derive a token sequence, it discovers a rightmost derivation in the reverse direction of a leftmost derivation

7

Parse tree of grammar



8

Handle and phrase

- A simple or prime phrase is a phrase that contains no smaller phase and is a sequence of symbols directly derived from a non-terminal
- The handle of a sentential form is the leftmost simple phrase
- Simple phrases cannot overlap ➔ leftmost simple phrase is unambiguous
- The sentential form F(V Tail), F and V Tail are simple phrases and F is the handle

9

Handle and phrase

- Handles are important because they represent individual derivation step, which can be recognized by various parsing techniques
- $\{[^i]^i \mid i>=1\ \}$ is not regular and can be generated by the CFG as followed:

  S➔ [T]

  T➔ [T] | λ

10

Handle and phrase

- Although CFGs are widely used to define the syntax of programming languages, not all syntactic rules are expressible using CFGs➔ variables must be declared before they are used cannot be expressed in a CFG and there is no way to transmit the exact set of variables that has been declared to the body of a program (static semantics and semantic routine with scope and type rules)

11

Linguistics

Type 0: Natural Language

Type 1: Context Sensitive Language ($\alpha A \beta \rightarrow \alpha \delta \beta$)

Type 2: Context Free Language($A \rightarrow \alpha \delta \beta$)

Type 3: Regular Language (No memory)

- Context sensitive and type 0 grammars are more powerful than CFG but they are far less useful ➔ the efficient parsers for these extended grammar classes does not exist just CFG has (Tradeoff)

12

Useless part of CFG production

CFG are a definitional mechanism and may have error

S→A | B
A→a
B→Bb → derived no terminal string (nonreduced)
C→c → unreachable so is useless (cannot from S)

Finally forms the reduced grammar

S→A
A→a

13

Ambiguous grammar

- A serious grammar flaw is that a grammar allows a program to have two or more different parse trees
- A grammar allows different parse trees for the same terminal strings
- Example

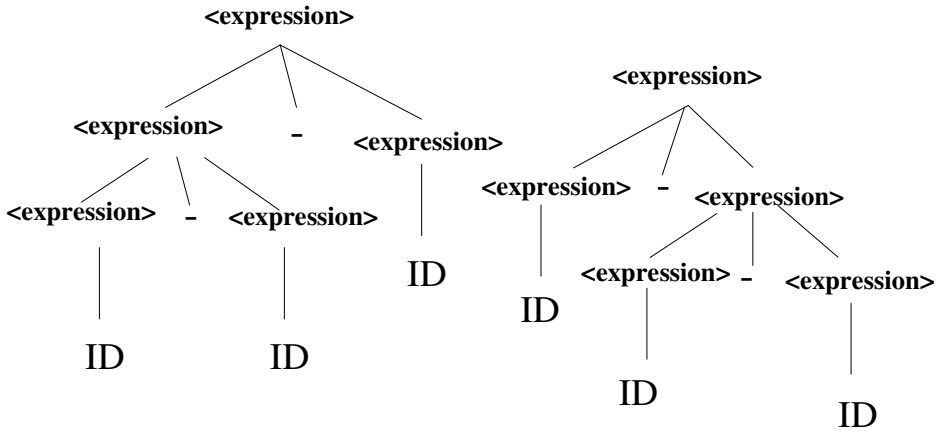
<exp> → <exp> - <exp> (left recursive)

<exp>→ID

14

Parse tree

There are 2 different parse trees for ID-ID-ID



15

Grammar notation

- A grammar can produce two parse trees is called ambiguous.
- Ambiguous grammar are rare used because a unique structure cannot be guaranteed for all inputs, and a unique translation, guided by the parse tree structure, may not be obtained
- Normally restrict to unambiguous grammars to guarantee unique structure
- Operator Precedence is a problem.
- Operator in an arithmetic expression is generated *lower* in the parse tree can be used to indicate that it *has precedence over* an operator produced *higher up* in the tree.

16

Grammar notation

<Assign> → <id> := <expr>
<id> → A<id_tail> | B<id_tail> | … | Z<id_tail>
<id_tail> → A<id_tail> | B<id_tail> | … | Z<id_tail> | 0<id_tail> | … | 9<id_tail> | λ
<expr> → <expr> <addop> <expr> | (<expr>) | <id>
<addop> → + | -

Syntactic ambiguity of language structures is a problem.

17

Grammar notation

A grammar can be written to separate the + and *
so they are consistently in a higher to lower ordering in the parse tree.

<Assign> → <id> := <expr>
< id > → A<id_tail> | B<id_tail> | … | Z<id_tail>
<id_tail> → A<id_tail> | B<id_tail> | … | Z<id_tail> | 0<id_tail> | … | 9<id_tail> | λ
<expr> → <expr> + <term> | <expr> − <term> | <term>
<term> → <term> * <factor> | <term> / <factor> | <factor>
<factor> → (<expr>) | <id>

18

Transforming EBNF to standard form

For (each production P=A→α **[x_1 … x_n]** β) {
  Create a new non-terminal, N. **Option**
  Replace production P with P'=A→α N β
  Add the productions: N→ x_1 … x_n and N→λ
}
For (each production Q=B→γ **{y_1 … y_m}**δ) {
  Create a new non-terminal, M. Option List
  Replace production Q with Q'=B→γ M δ
  Add the productions: M→ y_1 … y_mM and M→λ
}

19

CFG Important Rules

Rule 2. Eliminate the Left Recursion

CFG G=(V,T,S,P) , where P has the forms

$A \rightarrow Ax_1 \mid Ax_2 \mid Ax_3 \mid \ldots \mid Ax_n$

$A \rightarrow y_1 \mid y_2 \mid y_3 \mid \ldots \mid y_m$

Can be changed to

$A \rightarrow y_i \mid y_i Z$, i=1,2,…,m

$Z \rightarrow x_i \mid x_i Z$, i=1,2,…,m

Or

$A \rightarrow y_i Z$, i=1,2,…,m

$Z \rightarrow x_i Z \mid \varepsilon$, i=1,2,…,m

20

Parser and Recognizer

- Given an input string as a sequence of tokens, is this input syntactically valid? (can it be generated from the grammar?) ➔ a recognizer
- Is this input valid, what is its structure (parse tree)? An algorithm that answers this general question is a parser

21

Parser classification

- Top-down parser – it discovers the parse tree corresponding to a token sequence by starting at the top of the tree (start symbol) and then expanding it (via predictions) in a depth-first manner (LL ➔ leftmost)
- A top-down parse corresponds to a preorder traversal of the parse tree (中左右)
- Top-down parsing is predictive because they always predict the production that is to be matched before matching begins

22

Parser classification

- Bottom-up parser discovers the structure of a parse tree by beginning at its bottom (the leaves of the tree, terminal symbols) and determines the productions used to generate the leaves (LR➔rightmost)
- Then the productions used to generate the immediate parents of the leaves are discovered
- The parser continues until it reaches the production used to expand the start symbol
- A bottom-up parser corresponds to a postorder traversal of the parse tree (左右中)

23

Example grammar

<program> → **begin** <stmts> **end** $

<stmts> → <stmt>; <stmt>

<stmts> → λ

<stmt> → **simplestmt**

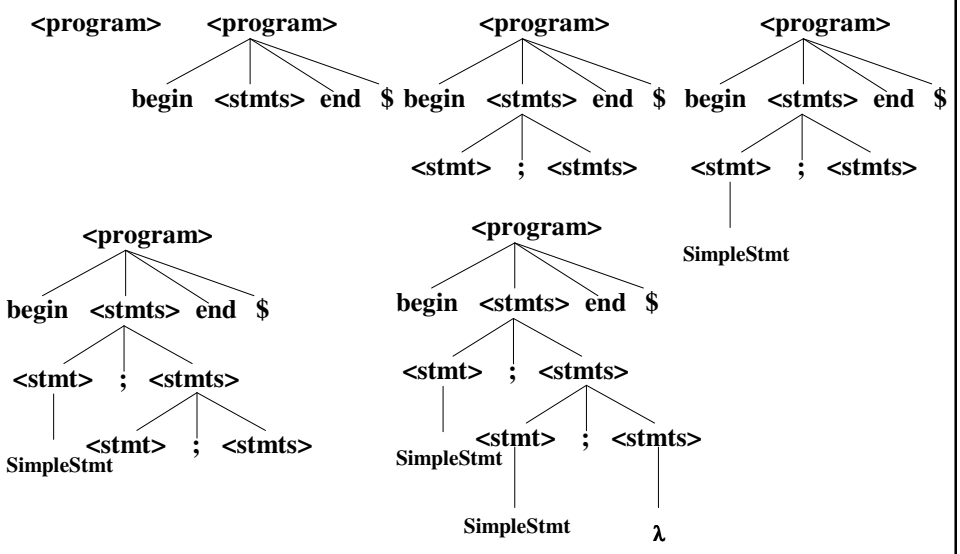
<stmt> → **begin** <stmts> **end**

For parsing the example

begin simplestmt; simplestmt; end$

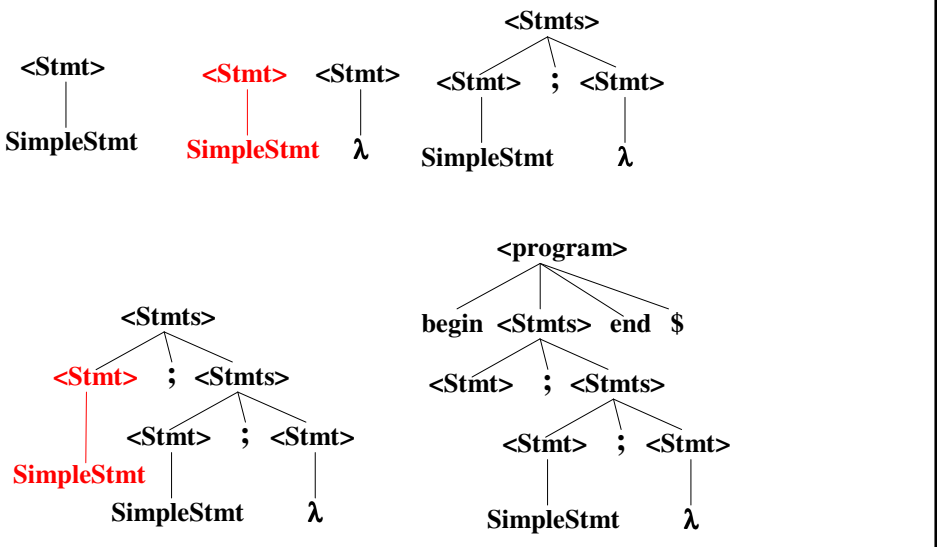
24

Top-down parse



25

Bottom-up parse



26

Explanation of parse tree

- A bottom-up parser proceeds by discovering sub-trees and linking them into increasingly larger trees
- A top-down parser represents a leftmost derivation and produces a leftmost parse
- A bottom-up parser represents a rightmost derivation and produces a rightmost parse
- The best known top-down and bottom-up parsing strategies are called LL and LR.

27

Explanation of parse tree

- The first L states that the token sequence will parse from left to right. The second letter (L or R) states whether a leftmost or rightmost parse will be produced
- We may characterize the parsing technique by including the number of lookahead symbols (symbols beyond the current token) the parser may use to make parsing. One-symbol lookahead is most common➔ LL(1) or LR(1)

28

CFG Important Rules

Rule 2. Eliminate the Left Recursion

CFG G=(V,T,S,P) , where P has the forms

$A \rightarrow Ax_1 \mid Ax_2 \mid Ax_3 \mid \ldots \mid Ax_n$

$A \rightarrow y_1 \mid y_2 \mid y_3 \mid \ldots \mid y_m$

Can be changed to

$A \rightarrow y_i \mid y_i Z$, i=1,2,…,m

$Z \rightarrow x_i \mid x_i Z$, i=1,2,…,m

Or

$A \rightarrow y_i Z$, i=1,2,…,m

$Z \rightarrow x_i Z \mid \varepsilon$, i=1,2,…,m

29

Example

$E \rightarrow E + T$

$E \rightarrow T$

Can be changed to

$E \rightarrow T \mid TZ$

$Z \rightarrow +T \mid +TZ$

Or

$A \rightarrow TZ$

$Z \rightarrow +TZ \mid \lambda$

30

Example

BNF: (left-recursive grammar)

<expr> →<expr> + <term> | <expr> - <term> | <term>

<term>→<term> * <factor> | <term> / <factor> | <factor>

EBNF: (non left-recursive grammar)

<expr>→<term> {(+ | -) <term>}

<term> →<factor> {(* | /) <factor>}

31

Example

<term> → <term> **(* | /) <factor>** | **<factor>**

By rule 2. Eliminate left-recursive

<term> → **<factor>** | **<factor>** <Z>

<Z> → **(* | /) <factor>** | **(* | /) <factor>** <Z>

By EBNF definition

<Z> → **(* | /) <factor>** { <Z> }

<Z> → **(* | /) <factor>** { **(* | /) <factor>** }

<Z> → { **(* | /) <factor>** }

By rule 5. <Z> is unit production

<term> → <factor> { <Z> }

<term> → <factor> { (* | /) <factor> }

32