Crafting a Compiler with C (VIII)

資科系
林偉川

The LL Grammar Class

- **Elimination of 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,…,n

Or

$A \rightarrow y_i A'$, i=1,2,…,m

$A' \rightarrow x_i A' \mid \lambda$, i=1,2,…,n

2

Grammar analysis algorithms

- It is necessary to analyze the properties of a grammar to determine if a grammar is readily parsable and to build the tables that can be used to drive a parsing algorithm
- Reinforce the basic concepts of grammars and derivations
- Many techniques that discussed previously need to build parsers are found as components of actual parser generators

3

Grammar Representation

- Vocabulary
  - Terminals
  - Nonterminals
- Productions
  - Field is an array of structs, each of which has fields
    - Lhs, rhs, rhs_length
- Start symbol

4

Grammar data structure

```
typedef int symbol, terminal, nonterminal;
#define VOCABULARY   (N_T+N_NONT)
typedef stru gram {
  int terminal[N_T], nonterminal[N_NONT];
  int start_symbol, num_productions;
  stru prod {
    int lhs, rhs_lenght, rhs[MAX_RHS];
  } productions[NUM_P];
  int vocabulary[VOCABULARY];
} grammar;
typedef stru prod production;
```

5

找出直接或間接推導成 λ

- One of the most common grammar computations is determining what non-terminal can derive λ (A➔ λ)
- Non-terminal can derive λ may disappear during a parse and must be carefully handled
- A➔BCD➔BC➔B➔ λ (may take more than one step)
  - Iterative marking algorithm
    - Non-terminals derive λ in one step are marked
    - Non-terminals requiring a parse tree height of two or more are found

6

找出直接或間接推導成 λ

- 透過演算法
  - Mark_lambda
  - Follow set

7

Determine if a Non-terminal can derive lambda

```
typedef short boolean;
typedef boolean marked_vocabulary[VOCABULARY];
marked_vocabulary mark_lambda(const grammar g){
  static marked_vocabulary derives_lambda;
  boolean changes, rhs_drives_lambda;
  int v, i, j;
  production p;
  for (v=0; v<VOCABULARY; v++)
     derives_lambda[v]=FALSE;
```

8

Determine if a Nonterminal can derive lambda

```
do {
  change=FALSE;
  for (i=0; i<g.num_productions; i++) {
    p=g.productions[i];
    if (! derives_lambda[p.lhs]) {
      if (p.rhs_length == 0) { // derives λ directly
        changes=derives_lambda[p.lhs]=TRUE;
        continue;
      }
      rhs_derives_lambda=derives_lambda[p.rhs[0]];
```

9

Determine if a Nonterminal can derive lambda

```
      for (j=1; j<p.rhs_length; j++)
        rhs_derives_lambda= rhs_derives_lambda
                          && derives_lambda[p.rhs[j]];
      if (rhs_derives_lambda) {
       changes-TRUE; derives_lambda[p.lhs]=TRUE;
      }
    }
  }
 } while (changes);
 return derives_lambda;
}
```

10

First set

- First(α) is the set of all the terminal symbols that can begin a sentential form derivable from α
- First(α)={ a ∈ V_t | α ➔*aβ }∪{if α ➔*λ then {λ} else ϕ }.
- If α is the right-hand side of a production, then First(α) contains terminal symbols that begin strings derivable from α
- First_set[X], and Follow_set[X] where X is a nonterminal. Elements of First and Follow set are terminals and λ

11

The LL Grammar Class

- **Def: FIRST(α) = { a | α =>* aβ } be the set of terminals derived from α (If α =>* λ, λ is in FIRST(α))**

  - *Pairwise Disjointness Test*:

  For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$

  ➔ For each non-terminal A, the first terminal symbol of its RHS must be unique!!!

12

The LL Grammar Class

1. A → a | bB | cAb
2. A → a | aB
First (A1) = { a, b, c }, First(A2) = { a }
FIRST(α_i) ∩ FIRST(α_j) = { a } ≠ ϕ ➔ not *Pairwise Disjointness*
Nonterminal A cannot be determined to use which of the RHS
Replace
<variable> → id | id [<expression>]
with
<variable> → id <new>, <new> → λ | [<expression>]
or
<variable> → id [[<expression>]]
(the outer brackets are meta-symbols of EBNF)

13

The LL Grammar Class

- **The Left Recursion Problem**
  If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser

  Ex. 1. Direct left recursive A → A + B
  2. Indirect left recursive A → B a A, B → Ab

- A grammar can be modified to remove left recursion
- Left recursive is a problem for all top-down parsing algorithm, but it is not a problem for bottom-up parsing algorithm (can eliminate the left recursive!!)

14

The LL Grammar Class

- The backtracking problem
  If a sequence of erroneous expansions and discover a mismatch, we have to undo the semantic effects of making these erroneous expansion. The recursive decent parser is the type of top-down parser that avoid backtracking.

15

The LL Grammar Class

- **Alternates problem**

Top-down backtracking parser is that the order in which alternates are tried can affect the language accepted.

Ex. S→cAd

A→ab | a ➔ cabd as the input string

if we choose A→ a instead of A→ab, we could fail to accepted!! (violate pairwise-disjoint rule)

16

The LL Grammar Class

- **The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness**
  The inability to determine the correct RHS on the basis of one token of lookahead ➔ First set test

17

Explanation of first_set algorithm

- For an arbitrary string α, compute_first(α) can return the set of terminals defined by FIRST(α)
- If α happens to be exactly one symbol long, compute_first(α) will simpy return first_set[α]
- Fill_first_set initializes first_set. The algorithm operates iteratively, first considering single productions, then considering chains of productions

18

Compute First(alpha)

```
typedef set_of_terminal_or_lambda termset;
termset follow_set[NUM_NONTERMINAL];
termset first_set[SYMBOL];
marked_vocabulary derives_lambda=mark_lambda(g);
termset  compute_first(string_of_symbols alpha) {
  int i,k;
  termset result;
  k=length(alpha);
```

19

Compute First(alpha)

```
  if (k == 0) result=SET_OF(λ);
  else {
   result=first_set[alpha[0]];
   for (i=1; i<k && λ ∈ first_set[alpha[i-1]]; i++)
     result=result ∪ (first_set[ alpha[i] ] - SET_OF(λ));
   if (i == k && λ ∈ first_set[ alpha[k-1] ])
     result=result ∪ SET_OF(λ);
  }
  return result;
}
```

20

Compute the First set

To compute First(X) for all grammar symbols X, we can have the following algorithm:

1. **If X is terminal, then FIRST(X) is { X }**
2. **If X is non-terminal and X → aα is a production, add a to FIRST(X). If X → λ is a production, then add λ to FIRST(X).**
3. **If X→$Y_1Y_2…Y_k$ is a production, then for all i ∋ $Y_1Y_2…Y_{i-1}$ are non-terminals and FIRST(Y_j) contains λ for j=1,2,…i-1 (i.e. $Y_1Y_2…Y_{i-1}$ =>* λ), add every non- λ symbol in FIRST(Y_j) to FIRST(X). If λ is in FIRST(Y_j) for j=1,2,…k, then add λ to FIRST(X).**

21

Compute First set for V

```
extern grammar g;
void fill_first_set(void) {
  nonterminal A;
  terminal a;
  production p;
  boolean changes;
  int i,j;
  for (i=0; i<NUM_NONTERMINAL; i++) { //1st loop
    A=g.nonterminals[i];
    if (derives_lambda[A]) first_set[A]=SET_OF(λ);
    else first_set[A]=ϕ;
  }
```

22

Compute First set for V

```
for (i=0; i<NUM_TERMINAL; i++) { // 2nd loop
  a=g.terminals[i];
  first_set[a]=SET_OF(a);
  for (j=0; j<NUM_NONTERMINAL; j++) {
    A=g.nonterminals[j];
    if (there exists a production A → aβ)
      first_set[A]=first_set[A] ∪ SET_OF(a);
  }
}
```

23

Compute First set for V

```
do { // 3rd loop
  change=FALSE;
  for (i=0; i < g.num_productions; i++) {
    p=g.productions[i];
    first_set[p.lhs]=first_set[p.lhs] ∪
                     compute_first[p.rhs];
    if (first_set changed) changes=TRUE;
  }
} while (changes);
}
```

24

Example of grammar

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

Step	First_set							
	E	Prefix	Tail	(	)	V	F	+
(1)First loop	ϕ	{λ}	{λ}					
(2)Second (nested) loop	{V}	{F,λ}	{+,λ}	{(}	{)}	{V}	{F}	{+}
(3)Third loop, production 1	{V,F,(}	{F,λ}	{+,λ}	{(}	{)}	{V}	{F}	{+}

25

Follow set

- When constructing parser, we often analyze a grammar to compute a set Follow(A), where A is any non-terminal.
- Follow(A) is the set of terminals that may follow A in some sentential form
- If A appears as the rightmost symbol in a sentential form, λ is include in Follow(A)
- Follow(A)={ a ∈ V_t | S➔+ …Aa }∪{if S➔+αA then {λ} else ϕ }

26

Follow set

- Follow(A) provides the lookahead that might signal the recognition of a production with A as the left-hand side
- **Def: FOLLOW(A) = { a | S =>* αAaβ for some α, β } be the set of terminals appear immediately to the right of A for some α, β. If A can be the rightmost symbol in some sentential form, then add λ to FOLLOW(A)**

27

Compute the Follow set

To compute Follow(A) for all non-terminals A, we can have the following algorithm:

1. **λ is in Follow(A), where A is the start symbol**
2. **If there is a production A→ αBβ , β ≠ λ, then everything in First(β) but λ is in Follow(B)**
3. **If there is a production A→ αB, or a production A→ αBβ where First(β) contains λ (i.e. β =>* λ), then everything in Follow(A) is in Follow(B)**

28

Compute Follow set for all Nonterminal

```
extern grammar g;
void fill_follow_set(void) {
  nonterminal A, B;
  boolean changes;
  int i;
  for (i=0; i<NUM_NONTERMINAL; i++) {
     A=g.nonterminals[i];  follow_set[A]=ϕ;
  }//initialization
 follow_set[g.start_symbol]=SET_OF(λ);
```

29

Compute Follow set for all Nonterminal

```
  do {
    change=FALSE;
    for (each production A → αBβ; i++) {
      follow_set[B]=follow_set[B] ∪
                        (compute_first[β]-SET_OF(λ));
       if (λ∈ compute_first(β))
           follow_set[B]=follow_set[B] ∪ follow_set[A];
       if (follow_set[B] changed) changes=TRUE;
      }
   } while (changes);
}
```

30

Example of grammar

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

Step	Follow_set		
	E	Prefix	Tail
(1)Initialization	{λ}	ϕ	ϕ
(2) Process Prefix in production 1	{λ}	{(}	ϕ
(3) Process E in production 1	{λ,)}	{(}	ϕ
(4) Process Tail in production 2	{λ,)}	{(}	{λ,)}

31

Example of grammar 1

S→aSe | B
B→bBe | C
C→cCe | d

Step	First_set							
	S	B	C	a	b	c	d	e
(1)First loop	ϕ	ϕ	ϕ					
(2)Second (nested) loop	{a}	{b}	{c,d}	{a}	{b}	{c}	{d}	{e}
(3)Third loop, production 2	{a,b}	{b}	{c,d}	{a}	{b}	{c}	{d}	{e}
(4)Third loop, production 4	{a,b}	{b,c,d}	{c,d}	{a}	{b}	{c}	{d}	{e}
(5)Third loop, production 2	{a,b,c,d}	{b,c,d}	{c,d}	{a}	{b}	{c}	{d}	{e}

32

Example of grammar 1

S→aSe | B
B→bBe | C
C→cCe | d

Step	Follow_set		
	S	B	C
(1)Initialization	{λ}	ϕ	ϕ
(2) Process S in production 1	{e,λ}	ϕ	ϕ
(3) Process B in production 2	{e,λ}	{e,λ}	ϕ
(4) Process B in production 3	{e,λ}	{e,λ}	ϕ
(5) Process C in production 4	{e,λ}	{e,λ}	{e,λ}
(6) Process C in production 5	{e,λ}	{e,λ}	{e,λ}

33

Example of grammar 2

S→ABc
A→a | λ
B→b | λ

Step	First_set					
	S	A	B	a	b	c
(1)First loop	ϕ	{λ}	{λ}			
(2)Second (nested) loop	ϕ	{a,λ}	{b,λ}	{a}	{b}	{c}
(3)Third loop, production 1	{a,b,c}	{a,λ}	{b,λ}	{a}	{b}	{c}

Step	Follow_set		
	S	A	B
(1)Initialization	{λ}	ϕ	ϕ
(2) Process A in production 1	{λ}	{b,c}	ϕ
(3) Process B in production 1	{λ}	{b,c}	{c}

34

Homework

S→AbB | d
A→CAb | B
B→cSd | λ
C→a | ed

<program> → **begin** <stmts> **end** $
<stmts> → <stmt>; <stmts>
<stmts> → λ
<stmt> → **simplestmt**
<stmt> → **begin** <stmts> **end**

<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>

35