Crafting a Compiler with C (III)

資科系
林偉川

Semantic specification

- Writing a complete and accurate compiler for a PL requires the language be well defined

Ex. In a if statement's boolean expression

(I != 0) && (k \ I > 10) , (I != 0) and (k \ I > 10) should be checked individually and then check the && operator. However, we can use a short-circuit evaluation for just if I == 0, then this expression is false ➔ this would formed a well defined way

2

PL that are easy to compile have advantages

- They are easier to learn, read, and understand
- They would have quality compilers on a wide variety of machines which would be crucial to a language's success
- Better code should be generated. Poor quality code can be fatal in critical application

3

PL that are easy to compile have advantages

- Fewer compiler bugs will occur
- Compiler will be smaller, cheaper, faster, more reliable, and more widely used
- Compiled diagnostics and program development features will often be better

4

Compiler v.s. PL design

- What attributes must be found in a PL to allow compilation
  - Can the scope and binding of each identifier reference be determined before execution (at compile time) ? ➔ Java compiler static type (strongly data type)
  - Can the type of an object be determined before execution? ➔ decide what type of operator ?
  - Can existing program text be changed or added to during execution? (code should be kept in some internal representation between source and machine code)

5

Compiler classification

- Diagnostic compiler
  - Aid in development and debugging of programs
  - Often repair minor errors automatically
  - Some program errors can be detected at runtime
    - Invalid script, misuse of pointer, illegal file manipulation
  - Have the capability to include checking code that can detect runtime error and abort program execution

6

Compiler classification

- Production compiler increases program speed by ignoring diagnostic concerns
- Optimizing compiler produces efficient machine code at the cost of increased compiler complexity and compilation time (how to just? Eliminate 1+0、v+0、v*1➔infeasible)

7

Compiler classification

- Optimizing compiler can produce optimal code for all programs. However, two programs are equivalent is un-decidable. Thus finding the simplest translation of a program cannot be done. Many program optimizations require time proportional to an exponential function of the size of the program being compiled
- Many optimizing compilers provide a number of levels of optimization, each providing greater code improvements at greater cost

8

Retarget-able compiler

- Target machine can be changed without rewriting its machine-independent components
- Difficult to write than an ordinary compiler because target machine dependencies must be carefully localized
- Difficult to generate code as efficient as that of an ordinary compiler because special cases and machine idiosyncrasies are harder to exploit
- Allow development cost to be shared and provides for uniformity across computers (important!!!)

9

Influence on compiler design

- Instruction sets have been non-uniform (register/memory operations)
- High-level operations have not been well supported (build-in stack and heap storage are hard to implement)
- Data and program integrity have been undervalued, and speed has been overemphasized
- Build-in function such as garbage collector, direct HW support and register & memory instruction classification
- RISC rather than CISC

10

Define the Micro compiler

- Just integer type
- The maximum length of identifier is 32 and begin with a letter composed of letter, digits and underscore
- Comment begin and end at the end of the current line

11

Define the Micro compiler

- Statement types are assignment(:=) and I/O
- Begin, end, read, and write are reserved words
- Each statement is terminated by a semicolon
- A source line is end with a blank and not extended

12

Structure of compiler

- One-pass compiler
- No explicit intermediate codes are used

13

Structure of compiler

- Compiler's components interface
  - Scanner reads a source program and produces tokens one at a time when called by the parser
  - Parser processes tokens until it recognizes a syntactic structure that requires semantic processing
  - Semantic routine produce output in assembly language for a simple 3-address virtual machine
  - No optimizer, code generation is done by direct calls to appropriate support routines from the semantics routines
  - Symbol table is used only by semantic routine

14

Token definition

```
typedef enum token_types {
  BEGIN, END, READ, WRITE, ID, INTLITERAL,
  LPAREN, RPAREN, SEMICOLON, COMMA,
  ASSIGNOP, PLUSOP, MINUSOP, SCANEOF
} token;

extern token scanner(void);
```

15

Construct a Micro scanner

- Scanner is a function of no arguments that return token value
- Scanner will read characters and group them into tokens
- Need to see the beginning of the next token in order to recognize the end of the current token
  ➔ one character of lookahead
- The extra character can be "pushed back" onto the input using "ungetc()" function

16

Construct a Micro scanner

- Assume that input can come from stdin; however, a **source file** opened by an explicit **FILE pointer** is better
- The scanner must find the beginning of some token ➔ first inspects the next input character. If the character cannot begin any token, we have a lexical or token error
- An error recover way is to skip the causing error character and restart scaning until the beginning of some token is found

17

Scanner loop for identifier and integer literal

```
#include <stdio.h>
#include <ctyoe.h>
int in_char, c;
while ((in_char =getchar()) != EOF()) {
  if (isspace(in_char)) continue;
  else if (isalpha(in_char)) { //ID::=letter | ID letter | ID digit | ID '-'
        for (c=getchar(); isalnum(c) || c == '-'; c=getchar()) ;
       ungetc(c,stdin); return ID;
  } else if (isdigit(in_char)) { //INTL::=digit | INTL digit
      while (isdigit(c=getchar())) ;
     ungetc(c,stdin); return INTLITERAL;
    } else lexical_error(in_char));
}
```

18

Scanner loop for operator, comments, delimiter

```
#include <stdio.h>
#include <ctyoe.h>
int in_char, c;
while ((in_char =getchar()) != EOF()) {
  if (isspace(in_char)) continue;
  else if (isalpha(in_char)) {
      for (c=getchar(); isalnum(c) || c == '-'; c=getchar()) ;
      ungetc(c,stdin); return ID; }
```

19

Scanner loop for operator, comments, delimiter

```
else if (isdigit(in_char)) {
   while (isdigit(c=getchar())) ;
   ungetc(c,stdin); return INTLITERAL; }
else if (in_char == '(') return LPAREN;
else if (in_char == ')') return RPAREN;
else if (in_char == ';') return SEMICOLON;
else if (in_char == ',') return COMMA;
else if (in_char == '+') return PLUSOP;
```

20

Scanner loop for operator, comments, delimiter

```
    else if (in_char == ':') { // looking for :=
      c=getchar();
       if (in_char == '=') return ASSIGNOP;
         else { ungetc(c, stdin); lexical_error(in_char)); }
     } else if (in_char == '-') { // looking for --, (comment)
         c=getchar();
         if (c == '-')  while ((in_char == getchar()) != '\n') ;
         else { ungetc(c, stdin); return MINUSOP; }
     } else lexical_error(in_char));
}
```

21

Scanner problem

- Not include the recognition of reserved words
  ➔ Homework!!!
- The reserved word problem looks the same as identifiers

22

Scanner problem for reserved word

- Method 1. ➔ Should have a reserved words table that is checked whenever an identifier is found
- If a token is found on this table, it is interpreted as a reserved word rather than as an identifier
- Method 2. ➔ Should modified the checking ID procedure to add check_reserved() to return ID or RESERVED

23

Explanation of Scanner algorithm

- Haven't made any provision for saving the characters of token as they are scanned
- Such as identifier and literals, we need the actual text of the token ➔ need token buffer
- **buffer_char()** adds its argument to character buffer called token_buffer
- clear_buffer() resets the buffer to the empty string
- token_buffer is visible to any part of the compiler and always contains the text of the most recently scanned token

24

Explanation of Scanner algorithm

- The characters in token_buffer will be used by check_reserved() to determine whether a token that looks like an identifier is actually a reserved word
- An end-of-file token called SCANEOF denoted by $ in formal description of parsing algorithms by parser generators

25

Complete version of part of Scanner

```
#include <stdio.h>
#include <ctyoe.h>
extern char token_buffer[];
token scanner(void) {
  int in_char, c;
  clear_buffer();
  if (feof(stdin)) return SCANEOF;
  while ((in_char =getchar()) != EOF()) {
    if (isspace(in_char)) continue;
    else if (isalpha(in_char)) {
      buffer_char(in_char);
      for (c=getchar(); isalnum(c) || c == '-'; c=getchar())
          buffer_char(c);
      ungetc(c,stdin); return check_reserved(); }
```

26

Complete version of part of Scanner

```
else if (isdigit(in_char)) {
  buffer_char(in_char);
  for (c=getchar(); isdigit(c); c=getchar())  buffer_char(c);
  ungetc(c,stdin); return INTLITERAL; }
else if (in_char == '(') return LPAREN;
else if (in_char == ')') return RPAREN;
else if (in_char == ';') return SEMICOLON;
else if (in_char == ',') return COMMA;
else if (in_char == '+') return PLUSOP;
```

27

Complete version of part of Scanner

```
else if (in_char == ':') { // looking for :=
  c=getchar();
   if (in_char == '=') return ASSIGNOP;
     else { ungetc(c, stdin); lexical_error(in_char)); }
  } else if (in_char == '-') { // looking for --, (comment)
      c=getchar();
      if (c == '-')
        do
          in_char=getchar();
        while (in_char != '\n') ;
      else { ungetc(c, stdin); return MINUSOP; }
  } else lexical_error(in_char));
 }
}
```

28

Example program for scanner

```
begin
  read (c1);
  c2:=c1+(5-c1);
  write (c2);
end$
```

Homework:

```
begin
  read c1,c3,c4;
  c2:=c1+(5-c3+c4);
  c3:=5+c2-c1;
  write c1,c2,c3,c4;
end$
```

29

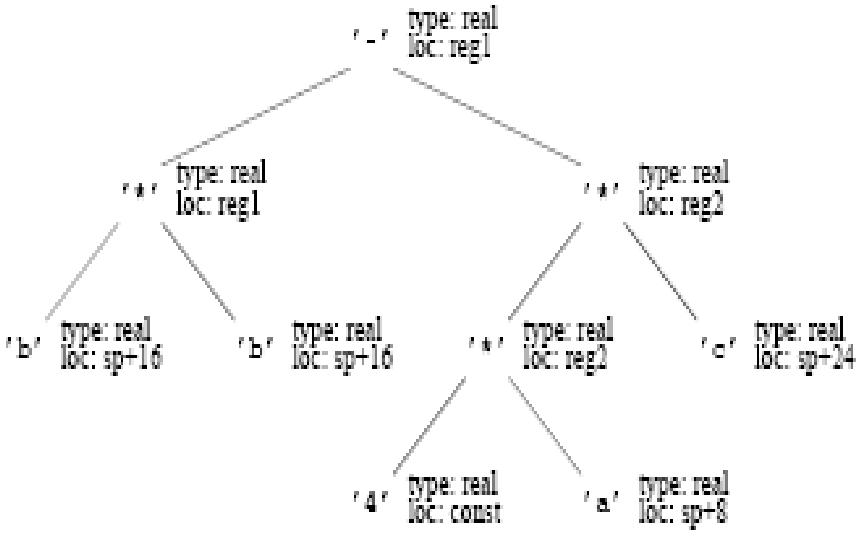
Symbol table

Index	String	Token_type
0	begin	BEGIN
1	read	READ
2	c1	ID
3	;	SEMICOLON
4	c2	ID
5	:=	ASSIGNOP
6	+	PLUSOP
7	(	LPAREN
8	5	INTLITERAL
9	-	MINUSOP
10	)	RPAREN

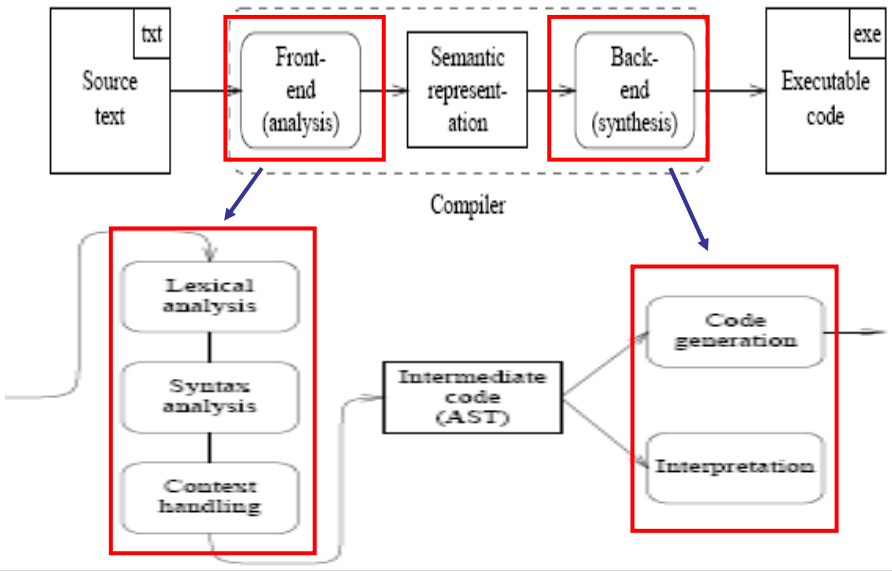
Index	String	Token_type
11	write	WRITE
12	end	END
13	$	SCANEOF

30

Annotated AST of an expression b*b-4*a*c



Structure of the compiler



Grammar for simply parenthesized expression

- expression → digit | ‘(‘ expression operator expression ‘)’
- operator → ‘+’ | ‘*’
- digit → ‘0’ | ‘1’ | … | ‘9’

- Fully parenthesized expression for producing 3,(5+8), (2*((3*4)+9))

33

Token types

- There are 5 different tokens such as DIGIT(0-9)、(、)、+、*
- We should split the expression into token of two parts, the class of token and its representation
- For token classes that contain only one token which is an ASCII character, the class is the ASCII value of the character itself
- The class of digits is DIGIT and defined as 257 and repr field is set to the representation of the digit

34

Token types

- The number over 255 are chosen to avoid collision with any ASCII values of single characters
- All tokens are single character, so a field of type char suffices
- A call to get_next_token() skips possible layout characters (‘ ‘,’\t’,’\n’) and stores the next single character as (class, repr) pair in Token
- A stream of tokens can be obtained by calling get_next_token() repeatedly

35

Data structure for symbol table(lex.h)

```
#define  EOF  256
#define  DIGIT  257
typedef struct {
  int class; char repre;
} Token_type;
extern Token_type Token;
extern void get_next_token(void);
```

36

Lexical analyzer for the demo compiler

```
#include “lex.h”
static int Layoutchar(int ch) {
  switch (ch) { // skip layout characters(‘ ‘,’\t’,’\n’)
    case ‘ ‘ :    case ‘\t’:   case ‘\n’:   return 1;
    default : return 0;
  }
}
```

37

Lexical analyzer for the demo compiler

```
Toke_type Token;
void get_next_token(void) {
  int ch;
  do { ch=getchar();
   if (ch<0) {
      Token.class=EOF; Token.repr=‘#’; return ; }
  } while (Layout_char(ch));
  if (‘0’ <= ch && ch <=‘9’) Token.class=DIGIT;
  else Token.class=ch;
  Token.repr=ch;
}
```

38

Syntax of Micro compiler

- CFG or BNF is used to define the syntax of Micro
- CFG is composed of 4 tuples {N, T, P, S}
- EBNF is used to enhance the BNF for optional items or lists of items ([ ] or { }) For example:

<program> → [ID:] begin <statement list> end

<statement list> → <statement> { <statement> }

<statement list> → <statement><statement tail>

<statement tail> → λ

<statement tail> → <statement><statement tail>

<statement list>→<statement>|<statement><statement list>

39

CFG Important Rules

Rule 4. Eliminate the λ production(Rule)

λ production:

$A \rightarrow \lambda$ or $A \Rightarrow^{*} \lambda$

Given a CFG G=(V,T,S,P) may generate a language not containing λ, the λ production can be removed

40

Example

G=({S,A},{a,b}, S, P), where P are given by

$S \rightarrow aAb$

$A \rightarrow aAb \mid \lambda$

can be changed to

$S \rightarrow aAb \mid ab$

$A \rightarrow aAb \mid ab$ ➔ $S \rightarrow aSb \mid ab$

Can generate the language $L=\{a^n b^n : n >= 1\}$

41

Extended CFG for Micro

Start: <system goal>➔ <program> SCANEOF

1.	<program>	→ **begin** <statement list> **end**
2.	<statement list>	→ <statement> {<statement>}
3.	<statement>	→ ID := <expression> ;
4.	<statement>	→ **read** (<id list>) ;
5.	<statement>	→ **write** (<expr list>) ;
6.	<id list>	→ ID {, ID}
7.	<expr list>	→ <expression> {, <expression>}
8.	<expression>	→ <primary> {<add op> <primary>}
9.	<primary>	→ (<expression>)
10.	<primary>	→ ID
11.	<primary>	→ INTLITERAL
12.	<add op>	→ PLUSOP
13.	<add op>	→ MINUSOP
14.	<system goal>	→ <program> SCANEOF

Figure 2.4 Extended CFG Defining Micro

Show example

For example:

How to show the following program is legal?

begin ID := ID + (INTLITERAL – ID); end

43

The derivation procedure

<program>	
begin <statement list> **end**	(Apply rule 1)
begin <statement> {<statement>} **end**	(Apply rule 2)
begin <statement> **end**	(Choose 0 repetition
begin ID := <expression> ; **end**	(Apply rule 3)
begin ID := <primary> {<add op> <primary>} ; **end**	(Apply rule 8)
begin ID := <primary> <add op> <primary> ; **end**	(Choose 1 repetition
begin ID := <primary> + <primary> ; **end**	(Apply rule 12)
begin ID := ID + <primary> ; **end**	(Apply rule 10)
begin ID := ID + (<expression>) ; **end**	(Apply rule 9)
begin ID := ID + (<primary> {<add op> <primary>}) ; **end**	(Apply rule 8)
begin ID := ID + (<primary> <add op> <primary>) ; **end**	(Choose 1 repetition
begin ID := ID + (<primary> – <primary>) ; **end**	(Apply rule 13)
begin ID := ID + (INTLITERAL – <primary>) ; **end**	(Apply rule 11)
begin ID := ID + (INTLITERAL – ID) ; **end**	(Apply rule 10)

44

CFG characteristics

- A CFG defines a language, which is a set of sequences of tokens
- Any sequence of tokens that can be derived using the grammar is valid
- Any sequence that cannot be derived is not valid
- Any token sequence derivable from a CFG is considered syntactically valid
- When static semantics are checked by the semantic routines, semantic errors in a syntactically valid program may be discovered.

  A:= 'X' + True; ➔ no syntax error but semantic error of '+' operator

45

Does CFG be ambiguous?

- Should consider two reason
  - Associativity (A-B-C)
  - Operator precedence (A+B*C)
- How to judge a CFG is ambiguous?
  - Derivation tree!!!

<expression> → <factor> { <add op><factor> }

<factor> → <primary> { <mult op><primary> }

<primary> → (<expression>) | ID | INTLITERAL

46

Associativity

- The order in which consecutive instances of an operator are applied

  For example: A-B-C or A+B-C or A*B/C

47

Operator precedence

- Refers to the relative priority of operators

  For example: A+B*C means A+(B*C)

  ➔ * is considered to be a higher precedence operator than +

  ➔ Micro does not support multiplication!!

Grammar defines such a precedence relationship

<expression> → <factor> {<add op><factor>}

<factor> → <primary> { <mult op><primary> }

<primary> → (<expression>) | ID | INTLITERAL

48

CFG for an expression

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (ID)$
$ID \rightarrow IDC \mid IDC\ IDN$
$IDC \rightarrow a \mid b \mid \ldots \mid z \mid a\ IDC \mid b\ IDC \mid \ldots \mid z\ IDC$
$IDN \rightarrow 0 \mid 1 \mid \ldots \mid 9 \mid 0\ IDN \mid 1\ IDN \mid \ldots \mid 9\ IDN$

49

Derivation Tree for A+B*C

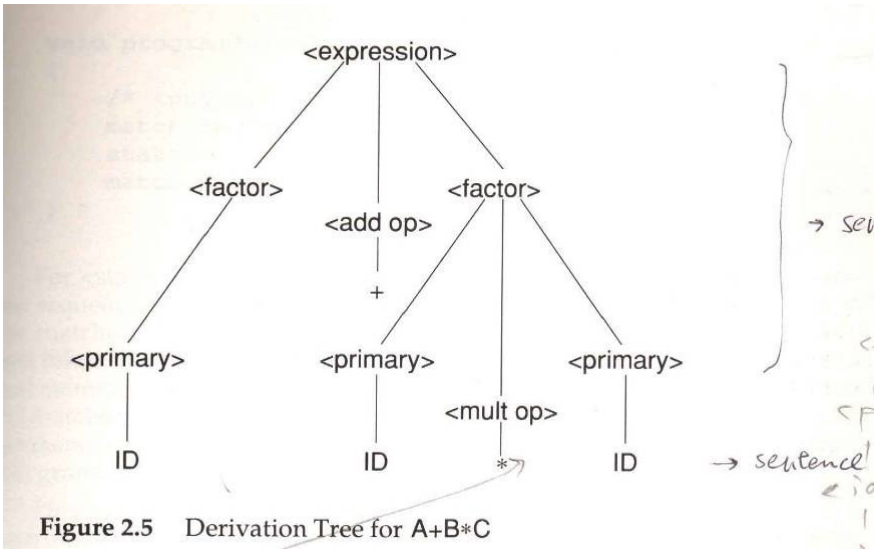


Figure 2.5 Derivation Tree for A+B*C

Derivation Tree for (A+B)*C

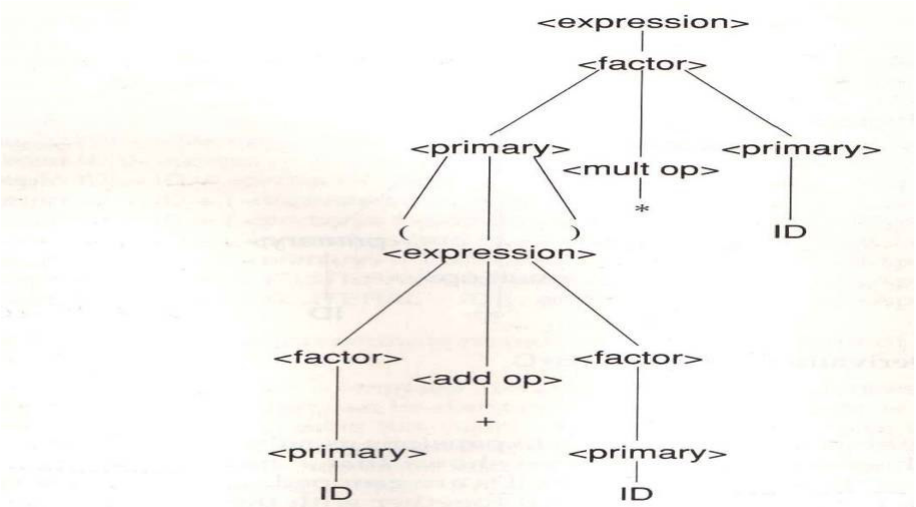


Figure 2.6 Derivation Tree for (A+B)*C