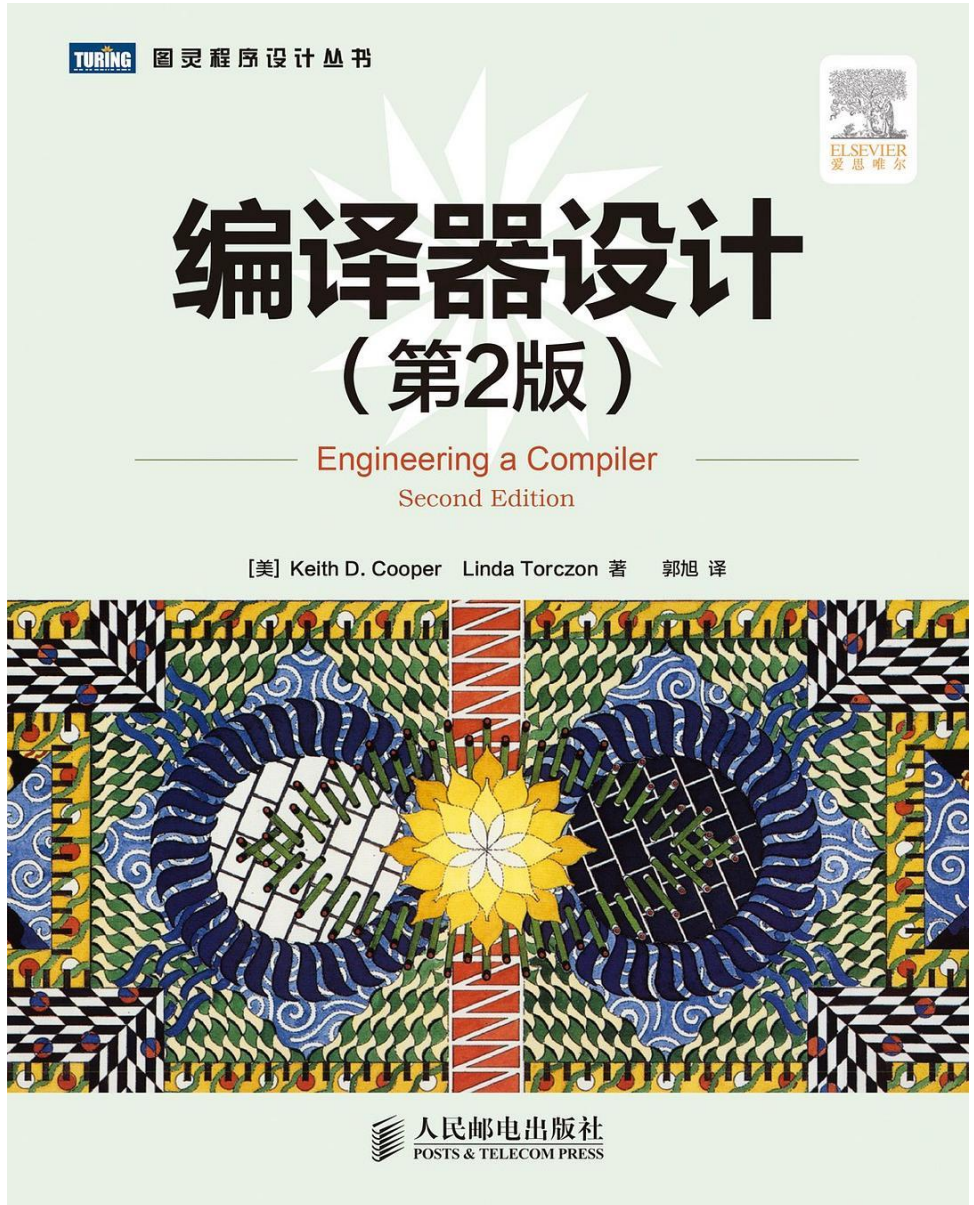
编译器入门

软件所智能软件中心PLCT实验室 王天然 实习生

2020/02/6



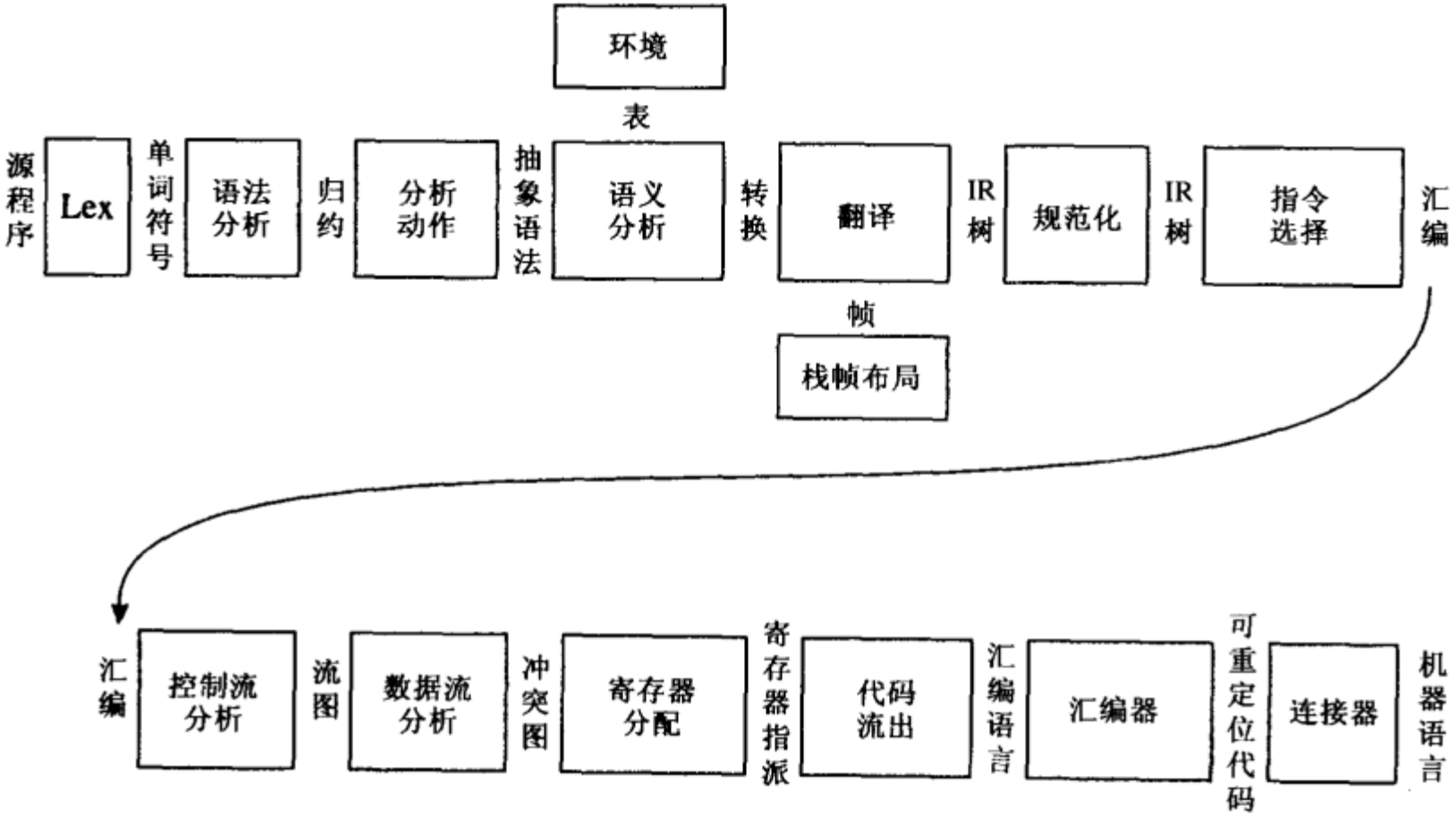
编译器设计(Keith D.Cooper)

现代编译原理C语言描述
(Andrew W.Appel)

目录

01 绪论

01 接口和模块

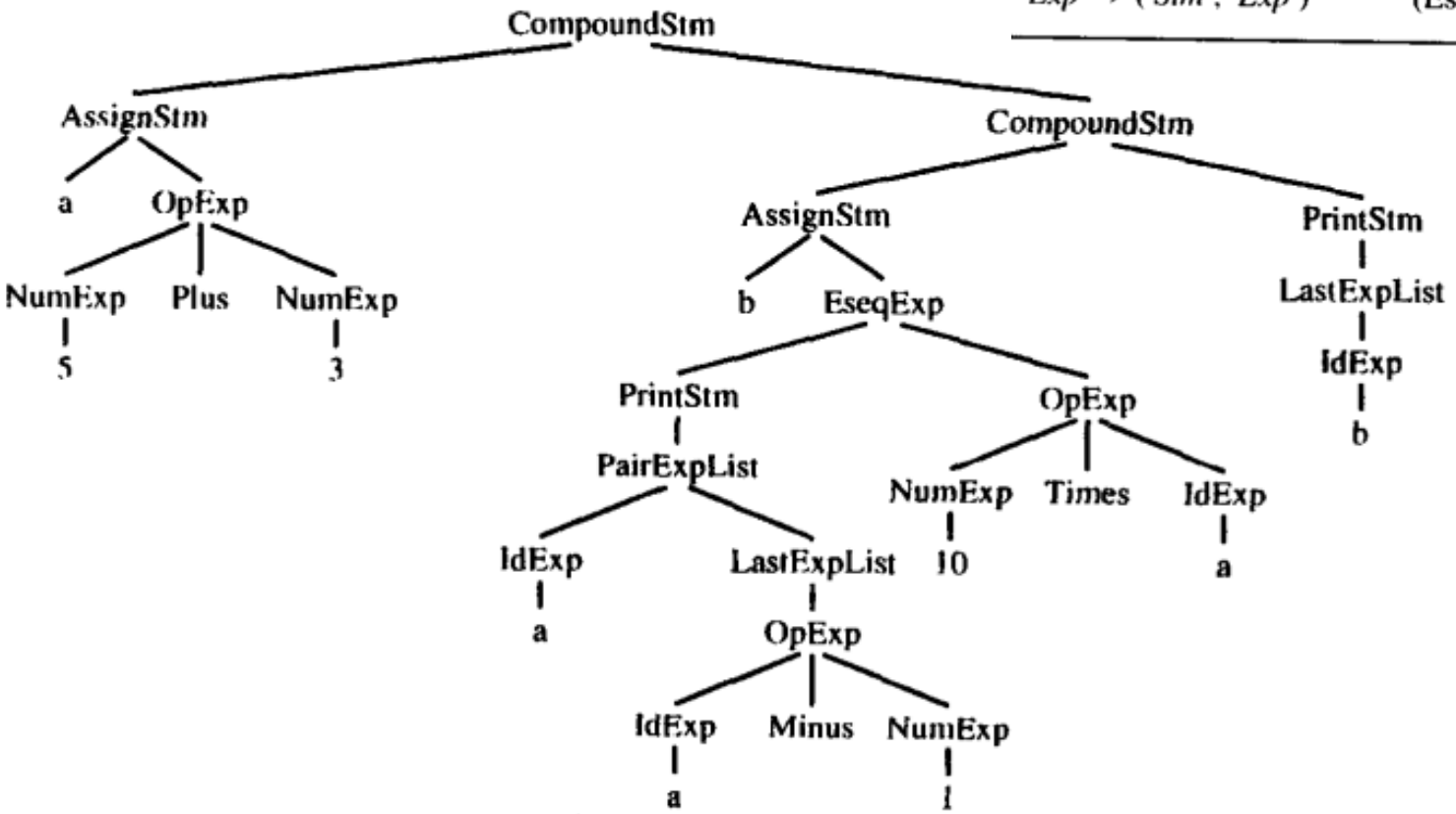


来源:现代编译原理C语言描述(Andrew W.Appel)

01 绪论

02 例子

Stm → *Stm* ; *Stm*	(CompoundStm)	*ExpList* → *Exp* , *ExpList*	(PairExpList)
Stm → `id :=` *Exp*	(AssignStm)	*ExpList* → *Exp*	(LastExpList)
Stm → `print (` *ExpList* `)`	(PrintStm)	*Binop* → +	(Plus)
Exp → `id`	(IdExp)	*Binop* → −	(Minus)
Exp → `num`	(NumExp)	*Binop* → ×	(Times)
Exp → *Exp Binop Exp*	(OpExp)	*Binop* → /	(Div)
Exp → `(` *Stm* , *Exp* `)`	(EseqExp)		




```
a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; print ( b )
```

来源:现代编译原理C语言描述(Andrew W.Appel)

02 词法分析

01 词法单词

词法分析器以字符流作为输入，生成一系列名字，关键字和标点符号，同时抛弃单词之间的空白符和注释。

词法单词是字符组成的序列，它可以看成是程序设计语言的文法单位，并且可以归为有限的几组单词类型。

```
ID        foo   n14   last
NUM       73   0 00   515   082
REAL      66.1   .5   10.   1e67   5.5e-10
IF        if
COMMA     ,
NOTEQ     !=
LPAREN    (
RPAREN    )
```

注释
预处理命令
预处理命令
宏
空格符、制表符和换行符

```
/* try again */
#include<stdio.h>
#define NUMS 5 , 6
NUMS
```


来源:现代编译原理C语言描述(Andrew W.Appel)

02 词法分析

01 词法单词

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.;
}
```

```
FLOAT   ID(match0)   LPAREN   CHAR   STAR   ID(s)   RPAREN
LBRACE   IF   LPAREN   BANG   ID(strncmp)   LPAREN   ID(s)
COMMA   STRING(0.0)   COMMA   NUM(3)   RPAREN   RPAREN
RETURN   REAL(0.0)   SEMI   RBRACE   EOF
```


来源:现代编译原理C语言描述(Andrew W.Appel)

02 正则表达式

为了得到一个简单可读性好的词法分析器。我们将用**正则表达式**的形式语言来指明词法单词，用**确定的有限自动机**来实现词法分析器，并用数学的方法将两者联系起来。

一种**语言**是字符串组成的集合，字符串是**符号**的有限序列。符号本身来自有限**字母表**。

来源:现代编译原理C语言描述(Andrew W.Appel)

02 词法分析

02 正则表达式

使用正则表达式用有限的描述来指明无限的语言。每个正则表达式代表一个字符串的集合。

- 符号(symbol) 可选(alternation) 联结(concatenation)

  ε(epsilon) 重复(repetition)

$(0\|1)^* \cdot 0$	由 2 的倍数组成的二进制数。
$b^*(abb^*)^*(a\|\epsilon)$	由 a 和 b 组成，但 a 不连续出现的字符串。
$(a\|b)^* aa(a\|b)^*$	由 a 和 b 组成，且有连续出现的 a 的字符串。

来源:现代编译原理C语言描述(Andrew W.Appel)

02 词法分析

02 正则表达式

词法分析器使用两条规则消除二义性

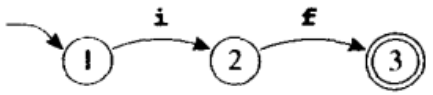
- **最长匹配**：初始输入子串中，取可与任何正则表达式匹配的那个最长的字符串作为下一个单词。**if8是一个标识符。**
- **规则优先**：对一个特定的最长初始子串，第一个与之匹配的正则表达式决定了这个子串的单词类型。**if是一个保留字。**

来源:现代编译原理C语言描述(Andrew W.Appel)

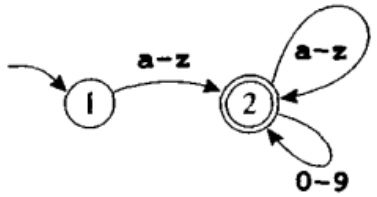
02 词法分析

03 有限自动机

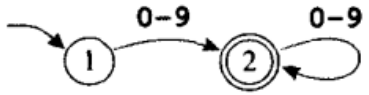
有限自动机有一个**有限状态集合**和一些从一个状态通向另一个状态的**边**，每条边上标记有一个**符号**；其中一个状态是初态，某些状态时终态。



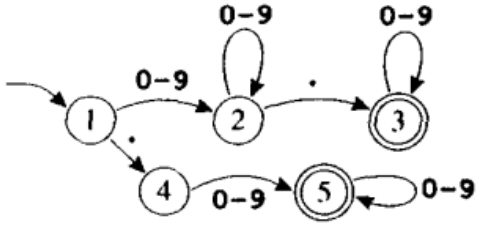
IF



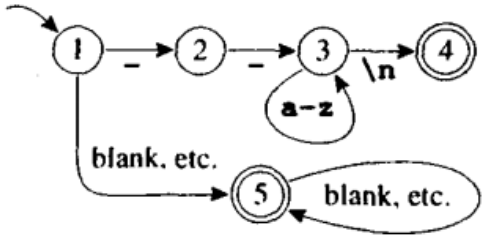
ID



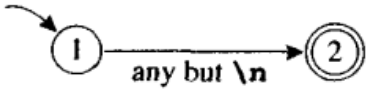
NUM



REAL



white space



error

来源:现代编译原理C语言描述(Andrew W.Appel)

04 非确定有限自动机

确定的有限自动机(DFA)不会有从同一状态出发的两条边标记有相同的符号。

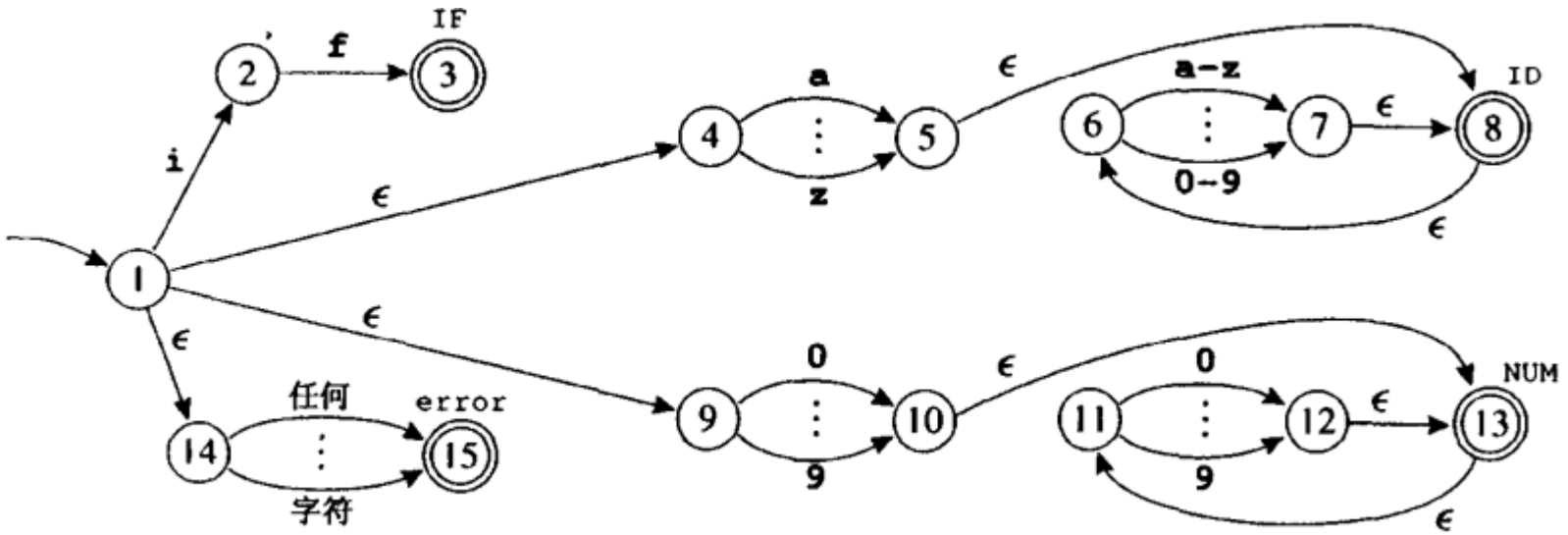
非确定有限自动机(NFA)是一种需要对从一个状态出发的多条标有相同符号的边进行选择的自动机。

可以很容易将一个正则表达式转换成一个NFA

来源:现代编译原理C语言描述(Andrew W.Appel)

02 词法分析

04 非确定有限自动机



来源:现代编译原理C语言描述(Andrew W.Appel)

04 Lex：词法分析器的生成器

```
%{
/* C Declarations: */
#include "tokens.h"    /* definitions of IF, ID, NUM, ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ   (EM_tokPos=charPos, charPos+=yyleng)
%}
/* Lex Definitions: */
digits   [0-9]+
%%
/* Regular Expressions and Actions: */
if                          {ADJ; return IF;}
[a-z][a-z0-9]*              {ADJ; yylval.sval=String(yytext);
                              return ID;}
{digits}                   {ADJ; yylval.ival=atoi(yytext);
                              return NUM;}
({digits}"."[0-9]*)|([0-9]*"."{digits})      {ADJ;
                              yylval.fval=atof(yytext);
                              return REAL;}
("--"[a-z]*"\n")|(" "|"\n"|"\t")+     {ADJ;}
                           {ADJ; EM_error("illegal character");}
```


来源:现代编译原理C语言描述(Andrew W.Appel)

03 语法分析

01 上下文无关文法

有限自动机缺少递归的表示方法。

上下文无关文法(context-free grammar)以说明的方式来定义语法。

1 $S \rightarrow S\ ;\ S$
2 $S \rightarrow \text{id} := E$
3 $S \rightarrow \text{print}\ (\ L\ )$

4 $E \rightarrow \text{id}$
5 $E \rightarrow \text{num}$
6 $E \rightarrow E + E$
7 $E \rightarrow (S\ ,\ E\ )$

8 $L \rightarrow E$
9 $L \rightarrow L\ ,\ E$

来源:现代编译原理C语言描述(Andrew W.Appel)

03 语法分析

01 上下文无关文法

推导有限自动机从开始符号出发对其右边的每一个非终结符，用非终结符对应的产生式中的任一右部来替换他

$\underline{S}$
S ; $\underline{S}$
$\underline{S}$; id := E
id := $\underline{E}$; id := E
id := num ; id := $\underline{E}$
id := num ; id := E + $\underline{E}$
id := num ; id := $\underline{E}$ + (S , E)
id := num ; id := id + ($\underline{S}$, E)
id := num ; id := id + (id := $\underline{E}$, E)
id := num ; id := id + (id := E + E , $\underline{E}$)
id := num ; id := id + (id := $\underline{E}$ + E , id)
id := num ; id := id + (id := num + $\underline{E}$, id)
id := num ; id := id + (id := num + num , id)

最左推导：一种总是扩展最左边非终结符的推导。

最右推导：一种总是扩展最右边非终结符的推导。

来源:现代编译原理C语言描述(Andrew W.Appel)

03 语法分析

02 预测分析

递归下降分析也称**预测分析，**适合于，每个子表达式的第一个终结符号能够为**产生式的选择**提供足够多的信息。

有时使用**语法分析器**生成工具并不方便，**预测分析器**的优点就在于其算法简单，可以用它手工构造分析器。

来源:现代编译原理C语言描述(Andrew W.Appel)

03 语法分析

02 预测分析

$S \rightarrow$ if E then S else S
$S \rightarrow$ begin S L
$S \rightarrow$ print E

$L \rightarrow$ end
$L \rightarrow$; S L

$E \rightarrow$ num = num

1. 构造预测分析表
2. 消除多重定义
3. 递归下降分析器

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);

enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();
                    eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();
       }}
void L(void) {switch(tok) {
        case END:   eat(END); break;
        case SEMI:  eat(SEMI); S(); L(); break;
        default:    error();
       }}
void E(void) {  eat(NUM); eat(EQ); eat(NUM); }
```


来源:现代编译原理C语言描述(Andrew W.Appel)

03 语法分析

02 预测分析

1. 构造预测分析表
2. 消除多重定义
3. 递归下降分析器

$S \rightarrow E\,\$$

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E'$
$E' \rightarrow -\,T\,E'$
$E' \rightarrow$

$T \rightarrow F\,T'$

$T' \rightarrow *\,F\,T'$
$T' \rightarrow /\,F\,T'$
$T' \rightarrow$

$F \rightarrow \mathrm{id}$
$F \rightarrow \mathrm{num}$
$F \rightarrow (\,E\,)$

	+	*	id	(	)	$
S			$S \rightarrow E\$$	$S \rightarrow E\$$		
E			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'	$E' \rightarrow +TE'$				$E' \rightarrow$	$E' \rightarrow$
T			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'	$T' \rightarrow$	$T' \rightarrow *FT'$			$T' \rightarrow$	$T' \rightarrow$
F			$F \rightarrow \mathrm{id}$	$F \rightarrow (E)$		

来源:现代编译原理C语言描述(Andrew W.Appel)

03 语法分析

03 LR分析

LR(k)代表从左至右分析、最右推导、超前查看k个单词

栈	输入	动作
	a := 7 ; b := c + (d := 5 + 6 , d) $	移进
id_4	:= 7 ; b := c + (d := 5 + 6 , d) $	移进
id_4 :=$_6$	7 ; b := c + (d := 5 + 6 , d) $	移进
id_4 :=$_6$ num_{10}	; b := c + (d := 5 + 6 , d) $	归约$E \rightarrow$ num
id_4 :=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	归约$S \rightarrow$ id:=E
S_2	; b := c + (d := 5 + 6 , d) $	移进
S_2 ;$_3$	b := c + (d := 5 + 6 , d) $	移进
S_2 ;$_3$ id_4	:= c + (d := 5 + 6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$	c + (d := 5 + 6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$ id_{20}	+ (d := 5 + 6 , d) $	归约$E \rightarrow$ id
S_2 ;$_3$ id_4 :=$_6$ E_{11}	+ (d := 5 + 6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$	(d := 5 + 6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8	d := 5 + 6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4	:= 5 + 6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$	5 + 6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ num_{10}	+ 6 , d) $	归约$E \rightarrow$ num
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11}	+ 6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11} +$_{16}$	6 , d) $	移进
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11} +$_{16}$ num_{10}	, d) $	归约$E \rightarrow$ num
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11} +$_{16}$ E_{17}	, d) $	归约$E \rightarrow E + E$
S_2 ;$_3$ id_4 :=$_6$ E_{11} +$_{16}$ (8 id_4 :=$_6$ E_{11}	, d) $	归约$S \rightarrow$ id:=E

来源:现代编译原理C语言描述(Andrew W.Appel)

03 语法分析

04 分析器生成器

1 $P \rightarrow L$

2 $S \rightarrow$ id := id
3 $S \rightarrow$ while id do S
4 $S \rightarrow$ begin L end
5 $S \rightarrow$ if id then S
6 $S \rightarrow$ if id then S else S

7 $L \rightarrow S$
8 $L \rightarrow L$; S

```
%{
int yylex(void);
void yyerror(char *s) { EM_error(EM_tokPos, "%s", s); }
%}
%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN
%start prog
%%

prog: stmlist

stm : ID ASSIGN ID
    | WHILE ID DO stm
    | BEGIN stmlist END
    | IF ID THEN stm
    | IF ID THEN stm ELSE stm

stmlist : stm
        | stmlist SEMI stm
```


来源:现代编译原理C语言描述(Andrew W.Appel)

04 上下文相关分析

01 概述

语法正确的输入程序仍然可能包含严重的错误，导致编译无法完成。为检测这样的错误，编译器需要进行更深层的检查，其中涉及每条语句放到实际的上下文中进行考虑。

为积累进一步转换需要的上下文知识，编译器必须开发出一些方法，从语法之外的视角来考察程序。

$$X \leftarrow Y, X \leftarrow Z$$

来源:编译器设计(Keith D.Cooper)

04 上下文相关分析

02 类型系统

大多数程序设计语言都将一组性质关联到每个数据值，这些性质的集合称为值的**类型**。与上下文无关语法相比利用**类型系统**可以在更精确的层次上规定程序的行为。

1. 确保运行时的安全性
2. 提高表达力
3. 生成更好的代码
4. 类型检查

来源:编译器设计(Keith D.Cooper)

04 上下文相关分析

03 属性语法

属性语法的是用于上下文相关分析的一种形式化机制。属性语法包括一个上下文无关语法，外加一组规定了某些计算的规则。每个规则都通过其他属性的值定义了一个值或属性。规则将属性关联到一个特定的语法符号，出现在语法分析树中的每个语法符号实例都有一个对应的属性实例。规则是功能性的，没有蕴涵特定的求值次序，且唯一地定义了每个属性值。

来源:编译器设计(Keith D.Cooper)

04 上下文相关分析

03 属性语法

$$
P = \left\{
\begin{array}{lcl}
Number & \rightarrow & Sign\ List \\
Sign & \rightarrow & + \\
 & | & - \\
List & \rightarrow & List\ Bit \\
 & | & Bit \\
Bit & \rightarrow & 0 \\
 & | & 1
\end{array}
\right\}
\qquad
\begin{array}{l}
T = \{+, -, 0, 1\} \\
NT = \{Number,\ Sign,\ List,\ Bit\} \\
S = \{Number\}
\end{array}
$$

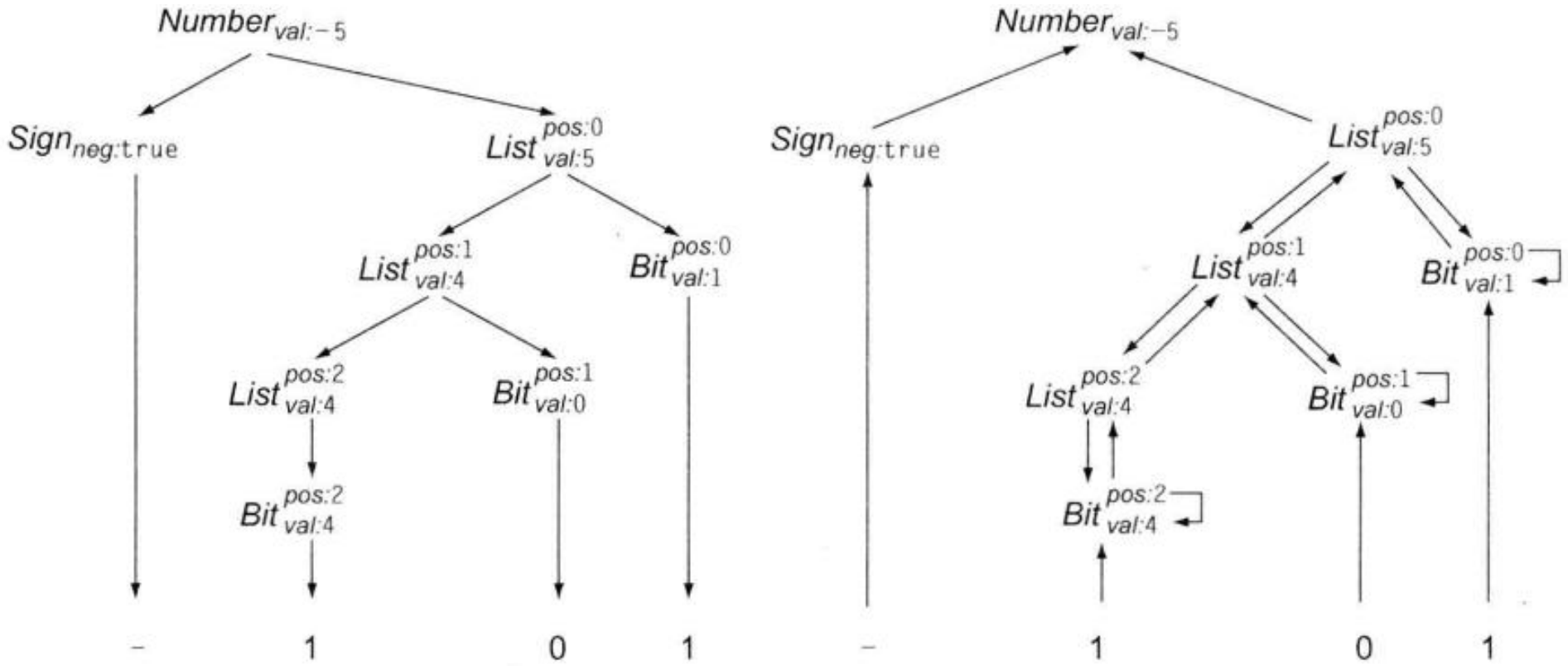
符号	属 性
Number	*value*
Sign	*negative*
List	*position, value*
Bit	*position, value*

	产 生 式	属性规则
1	*Number* → *Sign List*	*List.position* ← 0 *if Sign.negative* *then Number.value* ← -*List.value* *else Number.value* ← *List.value*
2	*Sign* → +	*Sign.negative* ← *false*
3	*Sign* → −	*Sign.negative* ← *true*
4	*List* → *Bit*	*Bit.position* ← *List.position* *List.value* ← *Bit.value*
5	$List_0 \rightarrow List_1\ Bit$	$List_1.position \leftarrow List_0.position + 1$ *Bit.position* ← $List_0.position$ $List_0.value \leftarrow List_1.value + Bit.value$
6	*Bit* → 0	*Bit.value* ← 0
7	*Bit* → 1	$Bit.value \leftarrow 2^{Bit.position}$

来源:编译器设计(Keith D.Cooper)

04 上下文相关分析

03 属性语法



(a) −101的语法分析树　　(b) −101的属性依赖关系图

图4-6　−101的属性化语法分析树

来源:编译器设计(Keith D.Cooper)

谢 谢

欢迎交流合作

2020/2/13