中山大學
SUN YAT-SEN UNIVERSITY

国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

DCS290

Compilation Principle

编 译 原 理

第五章 语法制导翻译（4）

郑馥丹
zhengfd5@mail.sysu.edu.cn

CONTENTS

目 录

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例： R → **addop** T { R1 .i = mknode(**addop**.lexeme, R.i, T.nptr); }

R1 { R.s = R1 .s; }

R → { R.s = R.i; }

– 只解析：

```
void R() {
    if (lookahead == addop) {
        match(addop);
        T();
        R()
    } else {
        // do nothing
    }
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 解析并翻译：

```
SyntaxTreeNode R(SyntaxTreeNode i) {
    SyntaxTreeNode s;  // synthesized attributes
    SyntaxTreeNode t_nptr, r1_i, r1_s;  // for children
    char addopLexeme;  // temporary

    if (lookahead == addop) {
        addopLexeme = lookahead.lexval;
        match(addop);
        t_nptr = T();
        r1_i = mknode(addopLexeme, i, t_nptr);
        r1_s = R(r1_i);
        s = r1_s;
    } else {
        s = i;
    }
    return s;
}
```

R → **addop** T { R1.i = mknode(**addop**.lexeme, R.i, T.nptr); }
    R1 { R.s = R1.s; }
R → ε { R.s = R.i; }

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 解析并翻译：

✓ 每个继承属性对应于一个形式参数（函数**A的参数**是非终结符**A的继承属性**）

✓ 所有综合属性对应返回值（函数**A的返回值**是非终结符**A的综合属性集合**）

- 多条综合属性可以合并到一条记录返回中

✓ 在函数A的函数体中，进行语法分析并处理属性：

- 决定用哪一个产生式来展开A
- 需读入某个**终结符**时，在输入中检查是否出现，若是，**match()**，否则报错
- 在局部变量中**保存所有必要的属性值**，这些值将用于计算产生式体中非终结符的继承属性，或产生式头的非终结符的综合属性
- **调用**产生式体**中非终结符的函数**，并提供参数

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：定点二进制小数转换为十进制小数

$N \rightarrow .\ \{ S.f = 1 \}\ S\ \{ print(S.v) \}$

$S \rightarrow \{ B.f = S.f \}\ B\ \{ S_1.f = S.f + 1 \}\ S_1\ \{ S.v = S_1.v + B.v \}$

$S \rightarrow \varepsilon\ \{ S.v = 0 \}$

$B \rightarrow 0\ \{ B.v = 0 \}$

$B \rightarrow 1\ \{ B.v = 2^{-B.f} \}$

例：$.1011=2^{-1}+0+2^{-3}+2^{-4}$

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：定点二进制小数转换为十进制小数

✓根据产生式：**N→ . { S.f = 1} S { print(S.v) }**

对非终结符N，构造如下函数：

```
void N()
{
    MatchToken('.');          //匹配'.'
    Sf =1;                        //变量 Sf 对应属性
S.f
    Sv = S(Sf);               //变量 Sv 对应属性S.v
    print(Sv);
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：定点二进制小数转换为十进制小数

✓根据产生式 **$S\rightarrow$ { B.f = S.f } B { S_1.f = S.f +1 } S1 {S.v = S_1.v + B.v }**

$S\rightarrow\varepsilon$ { S.v = 0 }

对非终结符S，构造如下函数：

```
float S(int f){
    if (lookahead=='0' or lookahead=='1'){
        Bf = f;   Bv = B(Bf); S1f = f+1;
        S1v = S(S1f); Sv = S1v + Bv;
    }
    else if (lookahead=='$')  Sv = 0;
    else { printf("syntax error \n"); exit(0); }
    return Sv;
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：定点二进制小数转换为十进制小数

✓根据产生式 **B→0 { B.v = 0 }**

B→1 { B.v = $2^{-B.f}$ }

对非终结符B，构造如下函数：

```
float B(int f){
    if (lookahead=='0'){ MatchToken('0'); Bv = 0 }
    else if(lookahead=='1'){
        MatchToken('1');    Bv = 2^(-f);
    }
    else{ printf("syntax error \n"); exit(0); }
    return Bv;
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

– 例：while循环语句文法 S → **while**(C)S_1

SDD

Productions	Semantic Actions

S → **while**(C)S_1	L1 = new(); //while语句开始位置
	L2 = new(); //S_1语句开始位置
	S_1.next = L1;
	C.false = S.next;
	C.true = L2;
	S.code = **label** || L1 || C.code || **label** || L2 || S_1.code;

SDT

S → **while**({ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; }
C) { S_1.next = L1; }
S_1 { S.code = **label** || L1 || C.code || **label** || L2 || S_1.code; }

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

S → **while**({ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; }
C) { S_1.next = L1; }
S_1 { S.code = **label** || L1 || C.code || **label** || L2 || S_1.code; }

存储并返回结果：

```
string S(label next){
    string Scode, Ccode; //存放代码片段的局部变量
    label L1,L2; //局部标号
    if(当前输入 == 词法单元while){
        读取输入;
        检查'('是下一个输入符号，并读取输入;
        L1 = new();
        L2 = new();
        Ccode = C(next,L2);
        检查')'是下一个输入符号，并读取输入;
        Scode = S(L1);
        return("label"||L1||Ccode||"label"||L2||Scode);
    }
    else … //其他语句类型
}
```

5. 在预测分析中实现L-属性定义

5）编写一个递归下降预测解析器（翻译器）

S → **while**({ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; print(“label”,L1); }
C) { S_1.next = L1; print(“label”,L2); }
S_1

边扫描边生成代码：

```
void S(label next){
    label L1,L2; //局部标号
    if(当前输入 == 词法单元while){
        读取输入;
        检查‘(’是下一个输入符号，并读取输入;
        L1 = new();
        L2 = new();
        print(“label”,L1);
        C(next,L2);
        检查‘)’是下一个输入符号，并读取输入;
        print(“label”,L2);
        S(L1);
    }
    else … //其他语句类型
}
```

随堂练习（3）

- 给定LL(1)文法G[E]及其翻译模式:

  E → T { R.in = T.val; } R { E.val = R.val; }

  R → + T { R1.in = R.in + T.val; } R1 { R.val = R1.val; }

  R → – T { R1.in = R.in – T.val; } R1 { R.val = R1.val; }

  R → ε { R.val = R.in; }

  T → num { T.val = num.val; }

  试对该翻译模式构造相应的递归下降预测翻译程序

随堂练习（3）

- 参考答案

E → T { R.in = T.val; } R { E.val = R.val; }
R → + T { R1.in = R.in + T.val; } R1 { R.val = R1.val; }
R → – T { R1.in = R.in – T.val; } R1 { R.val = R1.val; }
R → ε { R.val = R.in; }
T → num { T.val = num.val; }

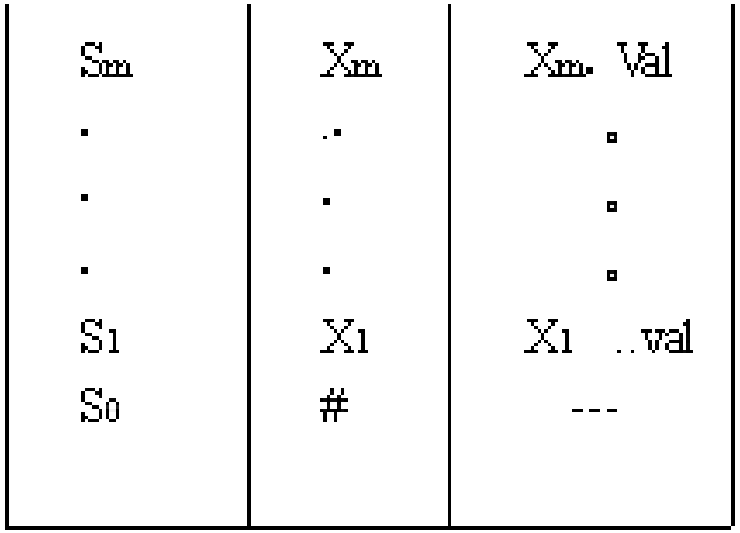
```
int E() {
    int t_val = T();
    int r_in = t_val;
    int r_val = R(r_in);
    return r_val;
}
```

```
int T() {
    if (isdigit(current_token.val) || current_token.val > 0) {
        int num_val = current_token.val;
        match(num_val);
        return num_val;
    } else {
        fprintf(stderr, "Syntax error: expected number\n");
        exit(1);
    }
}
```

```
int R(int in) {
    if (current_token.val == '+') {
        match('+');
        int t_val = T();
        int r1_in = in + t_val;
        int r1_val = R(r1_in);
        return r1_val;
    } else if (current_token.val == '-') {
        match('-');
        int t_val = T();
        int r1_in = in - t_val;
        int r1_val = R(r1_in);
        return r1_val;
    } else {
        // R → ε
        return in;
    }
}
```

6. 在LR分析中实现L-属性定义

- S-属性定义
  - 很容易进行LR分析——自底向上
  - 引入语义栈来存储属性值



- L-属性定义进行LR分析
  - 存在的挑战：
    - ✓ 若仅含综合属性：并不是所有语义动作都位于生产式体的最右边
    - ✓ 若含有继承属性：继承属性不存储在语义栈中
  - 应对策略：
    - ✓ 若仅含综合属性：**使用markers标记将所有嵌入的动作移到产生式体的最右边**
    - ✓ 若含有继承属性：**在语义栈中跟踪继承属性**

6. 在LR分析中实现L-属性定义

(1) 若仅含综合属性：使用markers标记将所有嵌入的动作移到产生式体的最右边

- marker可看成一个占位符
- marker替换掉L-属性SDT中的某个语义规则
- 对marker引入一条ε产生式，在产生式最右边附加原语义规则

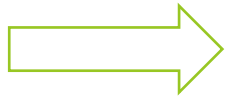
$E \rightarrow T\ R$

$R \rightarrow +\ T\ \{ print('+'); \}\ R$

$\quad | -\ T\ \{ print('-'); \}\ R$

$\quad | \varepsilon$

$T \rightarrow \mathbf{num}\ \{ print(\mathbf{num}.val); \}$



$E \rightarrow T\ R$

$R \rightarrow +\ T\ \mathbf{M}\ R$

$\quad | -\ T\ \mathbf{N}\ R$

$\quad | \varepsilon$

$T \rightarrow \mathbf{num}\ \{ print(\mathbf{num}.val); \}$

$\mathbf{M} \rightarrow \varepsilon\ \{ print('+'); \}$

$\mathbf{N} \rightarrow \varepsilon\ \{ print('-'); \}$

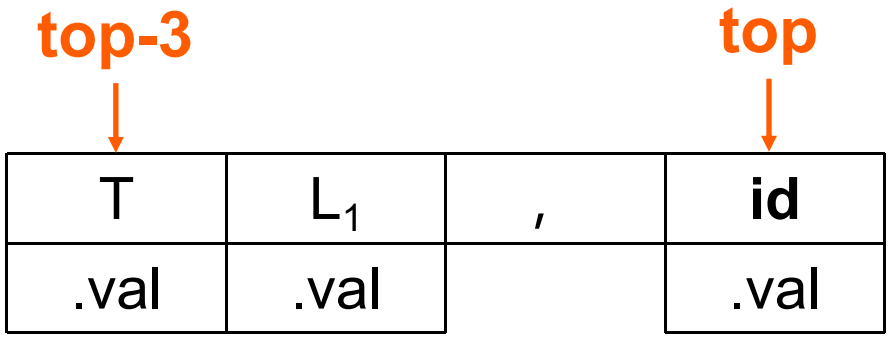
6. 在LR分析中实现L-属性定义

(2) 若含有继承属性：在语义栈中跟踪继承属性

– 最简单的情况：继承属性是通过复写规则(copy)从某个综合属性传播而来的

✓ 如：A→XYZ，将XYZ归约成A时，X的属性会先于Y的属性存在于语义栈中，若Y.i=X.s，则Y.i可直接从语义栈的X.s得来

D → T { L.inh = T.type; } L
T → **int** { T.type = INTEGER; }
T → **real** { T.type = REAL; }
L → { L_1.inh = L.inh; }
L_1 , **id** { addType(**id**.entry, L.inh); }
L → **id** { addType(**id**.entry, L.inh); }



Productions	Code
D → T L	
T → **int**	stack[ntop].val = INTEGER;
T → **real**	stack[ntop].val = REAL;
L → L_1 , **id**	addType(stack[top].val, stack[top – 3].val);
L → **id**	addType(stack[top].val, stack[top – 1].val);

6. 在LR分析中实现L-属性定义

(2) 若含有继承属性：在语义栈中跟踪继承属性

– 更复杂的情况1：若继承属性是通过普通函数而不是通过复写规则来定义的

✓ 如：S→aA {C.i=**f**(A.s)} C，在计算C.i时，A.s在语义栈上，但f(A.s)并未存在于语义栈，则：

- 引入新的marker **M**，将上述产生式规则改写为：

  S→aA **{M.i=A.s} M {C.i=M.s}** C

  M→ε {M.s=f(M.i)}

- 先执行**M→ε**的规则**{M.s=f(M.i)}**，算好f函数值M.s并保存在语义栈中后，才执行第一条产生式M后的规则**{C.i=M.s}**

6. 在LR分析中实现L-属性定义

(2) 若含有继承属性：在语义栈中跟踪继承属性

– 更复杂的情况2：若继承属性的访问存在不确定性

✓ 如：S→aA {C.i=A.s} C | bAB {C.i=A.s} C　　C→c {C.s=g(C.i)}

在使用C→c进行归约时，不确定C.i应该使用语义栈top-1位置的，还是语义栈top-2位置的，则：

- 引入新的marker **M**，将上述产生式规则改写为：

  S→aA {C.i=A.s} C | bAB **{M.i=A.s} M {C.i=M.s}** C

  M→ε {M.s=M.i}

  C→c {C.s=g(C.i)}

- 此时在使用C→c进行归约时，C.i的值就确定地可以通过访问语义栈top-1而得

6. 在LR分析中实现L-属性定义

- 例：while循环语句文法 S → **while**(C)S1

SDT

S → **while**({ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; }
C) { S_1.next = L1; }
S_1 { S.code = **label** || L1 || C.code || **label** || L2 || S_1.code; }

S → **while**(**M** C) **N** S_1 **{ 语义代码3 }**
M → ε **{ 语义代码1 }**
N → ε **{ 语义代码2 }**

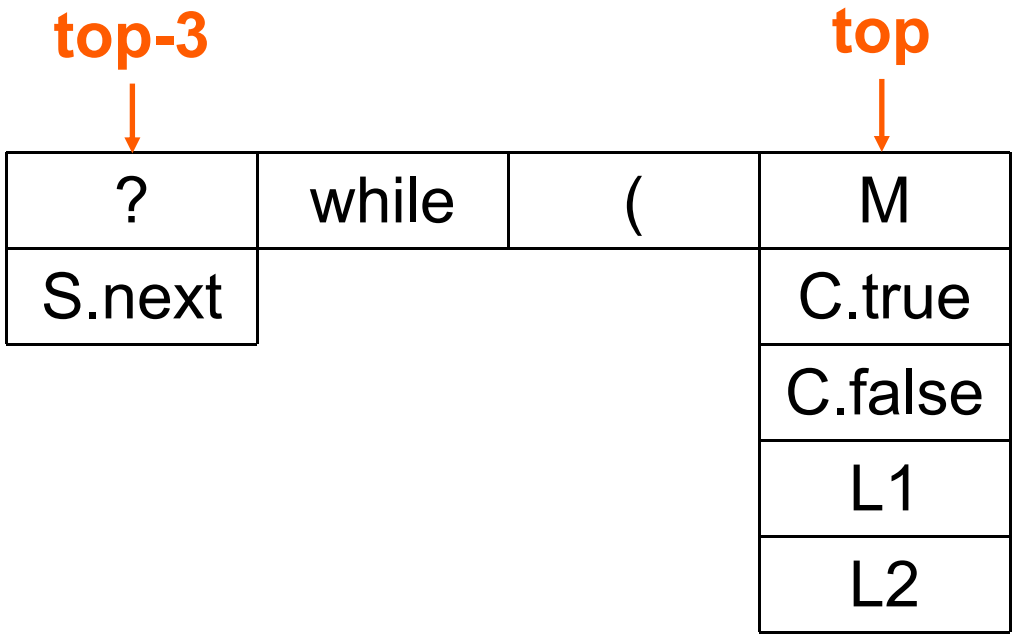
语义代码1： L1 = new(); L2 = new(); C.false = S.next; C.true = L2;
语义代码2： S_1.next = L1;
语义代码3： S.code = **label** || L1 || C.code || **label** || L2 || S_1.code;

6. 在LR分析中实现L-属性定义

- 例：while循环语句文法 S → **while**(C)S1

S → **while**(**M** C) **N** S_1 **{ 语义代码3 }**
M → ε **{ 语义代码1 }**
N → ε **{ 语义代码2 }**

语义代码1：
L1 = new(); L2 = new();
C.false = S.next; C.true = L2;



语义代码1：
L1 = new();
L2 = new();
C.true = L2;
C.false = stack[top-3].next;

6. 在LR分析中实现L-属性定义

- 例：while循环语句文法 S → **while**(C)S1

S → **while**(**M** C) **N** S_1 **{ 语义代码3 }**
M → ε **{ 语义代码1 }**
N → ε **{ 语义代码2 }**

语义代码2： S_1.next = L1;

top-3 **top**

?	while	(	M	C	)	N
S.next			C.true	C.code		S_1.next
			C.false			
			L1			
			L2			

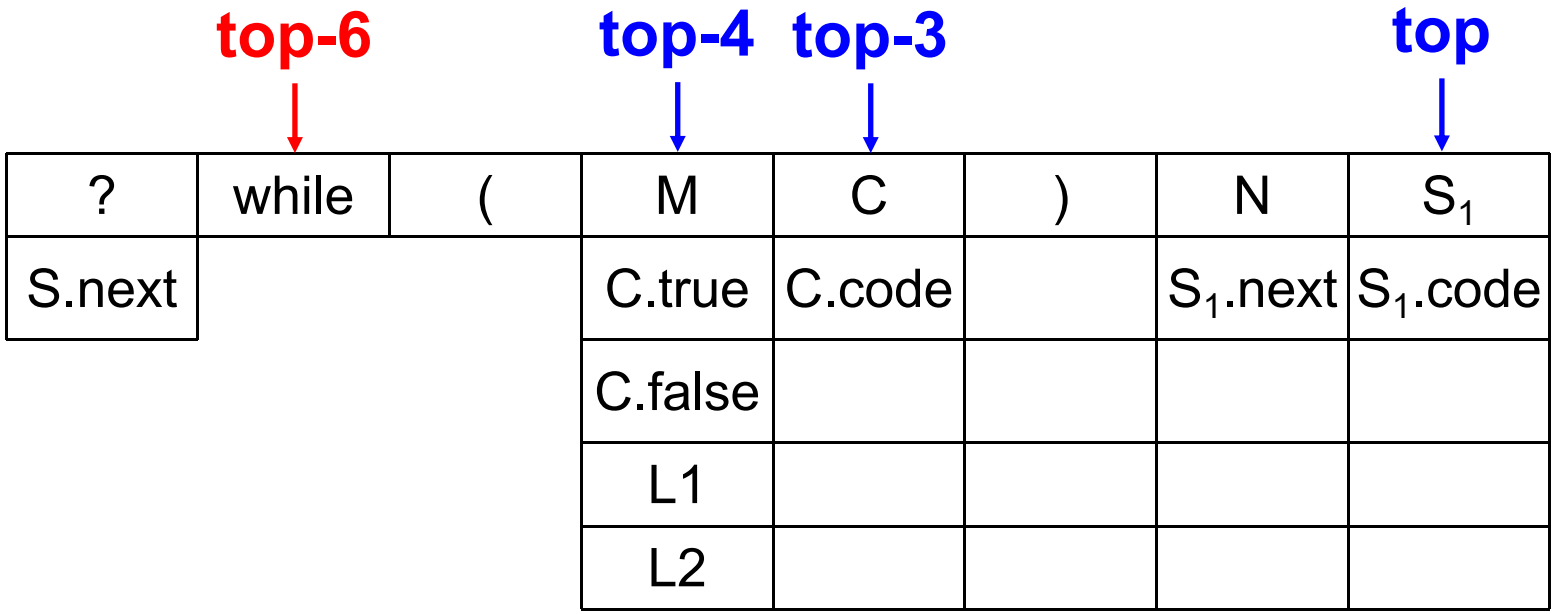
语义代码2：
S_1.next = stack[top-3].L1;

6. 在LR分析中实现L-属性定义

- 例：while循环语句文法 S → **while**(C)S1

S → **while**(**M** C) **N** S_1 **{ 语义代码3 }**
M → ε **{ 语义代码1 }**
N → ε **{ 语义代码2 }**

语义代码3：
S.code = **label** || L1 || C.code ||
label || L2 || S_1.code;



语义代码3：
tempCode = **label** ||stack[top-4].L1 || stack[top-3].code || **label** || stack[top-4].L2 || stack[top].code;
top=top-6;
stack[top].code=tempCode;

第五章作业

- 给定LL(1)文法G[S]及其翻译模式：

  S → Ab {B.in_num=A.num} B {if B.num=0 then print(“Accepted!”) else print(“Refused!”)}

  A → a A_1 {A.num=A_1.num+1}

  A → ε {A.num=0}

  B → a {B_1.in_num=B.in_num} B_1 {B.num=B_1.num-1}

  B → b {B_1.in_num=B.in_num} B_1 {B.num=B_1.num}

  B → ε {B.num=B.in_num}

  试对该翻译模式构造相应的递归下降预测翻译程序

第五章作业

- 提交要求：
  - 文件命名：学号-姓名-第五章作业；
  - 文件格式：.pdf文件；
  - 手写版、电子版均可；若为手写版，则拍照后转成pdf提交，但**须注意将照片旋转为正常角度，且去除照片中的多余信息**；电子版如word等转成pdf提交；
  - 提交到超算习堂（第五章作业）处；
  - 提交ddl：**4月29日晚上12:00**；
  - **重要提示：不得抄袭！**