

Construção de Compiladores

Riverson Rios, Ph.D.

Curso de Bacharelado em Computação
DC/UFC
2005



Riv Rios DC/UFC

04/09/99

1



Programa

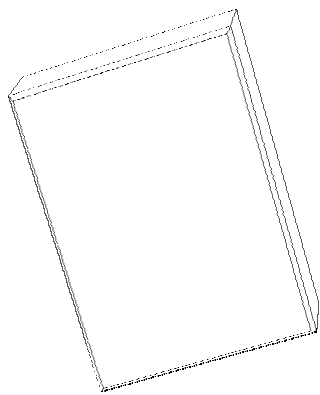
1. Introdução à Compilação
2. Análise Léxica
3. Análise Sintática
4. Análise Semântica
5. Geração de Código
6. Otimização de Código

Riv Rios DC/UFC

04/09/99

2

Bibliografia Recomendada



(Livro do Dragão)

Aho, A.V., Sethi, R. & Ullman,
J.D. (1995). *Compiladores –
Princípios, Técnicas e
Ferramentas*. Guanabara
Koogan

Riv Rios DC/UFC

04/09/99

3

Bibliografia Complementar

- Aho & Ullman (72). *The Theory of Parsing, Translation and Compiling. Vol. I: Parsing*. Prentice-Hall
- Aho & Ullman (72) ----- *Vol. II: Compiling*. Prentice-Hall
- Appel, A. (02). *Modern Compiler Implementation in Java*. 2nd Ed, Cambridge University Press
- Grune, Bal, Jacobs, Langendoen (01). *Modern Compiler Design*. Wiley
- José Neto (87) *Introdução à Compilação*. LTC
- Kowaltowski (83) *Implementação de Linguagens de Programação*. Guanabara
- Setzer & Melo (83) *A Construção de um Compilador*. Campus
- Tremblay (81) *Compiler Writing: Theory and Practice*. Mc Graw-Hill
- Pratt (96) *Programming Languages: Design and Implementation*. 3rd ed. Prentice Hall

Riv Rios DC/UFC

04/09/99

4

1. Introdução à Compilação

2. Análise Léxica

3. Análise Sintática

4. Análise Semântica

5. Geração de Código

6. Otimização de Código



1.1 Compiladores

- Def.: tradutor de linguagem-fonte para linguagem-alvo, reconhecendo a gramática fonte e indicando possíveis erros

Compilador = reconhecedor
+ gerador de código

- Notação T
 - Ex. Construção de um compilador para uma máquina recém construída

1.1.1 Tipos

- Tradutor de LAN \Rightarrow LO (compilador)
- Tradutor de LAN \Rightarrow LM (*assembler*)
- Tradutor de LAN \Rightarrow LAN
- Tradutor de LN \Rightarrow LN
- Tradutor de LHT \Rightarrow LHT
- Tradutor de LO \Rightarrow LAN

1.1.2 Interpretação

- Simulação por *software*
- Execução do programa em LA feita em outro computador
- Instruções são decodificadas e executadas uma a uma. Algumas podem nunca ser alcançadas
- Vantagem: pouco espaço
- Desvantagem: tempo de execução
- LPs interpretadas: Lisp, Snobol, Logo, APL

1.2 O processo de compilação

- Analisador Léxico
- Analisador Sintático
- Analisador Semântico
- Gerador de Código Intermediário
- Otimizador de Código Intermediário
- Gerador de Código
- Otimizador de Código
- Tabela de Símbolos



1.2.1 Modelos de implementação

- A função principal é o:
 - Analisador Léxico
 - Analisador Sintático
 - Gerador de Código
- Co-rotinas

Exercícios:

3.1, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8 (Dragão)

1. Introdução à Compilação

2. Análise Léxica

3. Análise Sintática

4. Análise Semântica

5. Geração de Código

6. Otimização de Código

2.1 Funções

Identificar *tokens* no texto-fonte

Guardar numeração das linhas

Gerar referência cruzada

Conversão de constantes

Ignorar alguns elementos sintáticos

Inserir identificadores (ids) na tabela de símbolos

2.1.1 Token

Lexema

“soma”, “:=”, “to”, “200”, “<=” &c.

Classes

Operador relacional

Palavra reservada

Fecha-parêntese

Abre-parêntese

Identificador

Inteiro

&c



2.2 Linguagens

Def.: Conjunto de *strings* formados a partir de elementos de um alfabeto Σ

Exemplos para $\Sigma = \{a,b\}$

L1 = {ab, ba}

L2 = {a, aa, aaa, aaaa, ...}

L3 = { ϵ , a, aa, aaa, aaaa, aaaaa, ...}

L4 = {}

L5 = { ϵ }

2.2.1 Operações com Linguagens

Operações:

União

Concatenação

Fechamento

Fechamento Positivo

Aplicações

2.2.2 Expressões Regulares

Def. (recursiva):

- ϵ é uma e.r.
- Se $\alpha \in \Sigma$, α e (α) são e.r.
- Se α e β são e.r., então $\alpha\beta$ é uma e.r., denotando a linguagem $L(\alpha)L(\beta)$
- Se α e β são e.r., então $\alpha|\beta$ é uma e.r., denotando a linguagem $L(\alpha) \cup L(\beta)$
- Se α é uma e.r., então α^+ é uma e.r., denotando a linguagem $L(\alpha)^+$
- Se α é uma e.r., então α^* é uma e.r., denotando a linguagem $L(\alpha)^*$
- Nada mais é e.r.

2.2.3 Aplicações de expressões regulares

Definição de constantes inteiras

- $d = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
- $n = d^+$
- $i = (+|-|\epsilon)n$
- $r = \dots$

Ids do Cobol

- $l = a | b | \dots | z | A | B | \dots | Z$
- $id = \dots$

Ids do Fortran



2.3 Autômatos

Def.: reconhecedores de linguagens a partir da transição de estados

Tipos:

determinísticos (DFAs) - o próximo passo é conhecido, independentemente do estado corrente

não-determinísticos (NFAs) - não se sabe o próximo passo pois possivelmente existe mais de uma alternativa

2.3.1 AFs determinísticos

DFA = $\langle Q, \Sigma, \delta, Q_0, F \rangle$ onde:

Q - conjunto de estados

Σ - alfabeto

δ - função TOTAL de transição

$\delta: Q \times \Sigma \Rightarrow Q$

Q_0 - estado inicial

$Q_0 \subseteq Q$

F - estados finais

$F \subseteq Q$

2.3.2 AFs não-determinísticos

NFA = $\langle Q, \Sigma, \delta, Q_0, F \rangle$ onde:

Q - conjunto de estados

Σ - alfabeto

δ - função TOTAL de transição

$\delta: Q \times (\Sigma \cup \epsilon) \Rightarrow P(Q)$

P é o conjunto das partes

Q_0 - estado inicial, $Q_0 \subseteq Q$

F - estados finais, $F \subseteq Q$

2.4 Equivalência

a) entre DFAs e NFAs

DFA \Rightarrow NFA?

Sim. Nada a fazer pois todo DFA já é um NFA!

NFA \Rightarrow DFA

Sim. Por construção, basta definir um novo conjunto composto dos subconjuntos do conjunto de estados e estender a função de transição.



b) e.r. \Rightarrow NFA?

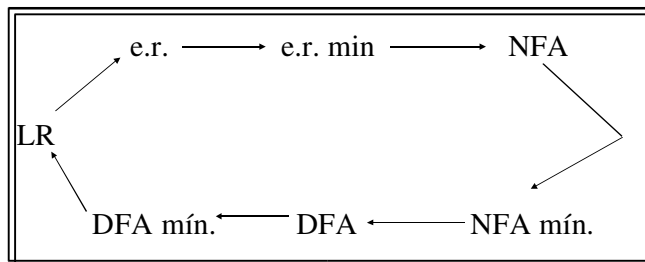
Sim. Por construção, basta criar um novo NFA para cada operador da e.r., usando e abusando de transições- ϵ e depois ligá-los, mais uma vez utilizando transições- ϵ , de acordo com a e.r. O NFA final pode posteriormente ser reduzido.

c) N.FA. \Rightarrow NFA mínimo?

d) D.FA. \Rightarrow DFA mínimo?

Sim. Existem algoritmos para a construção de NFAs e DFAs mínimos (e únicos). É possível, ainda, definirem-se a e.r. mínima para uma dada e.r.

Resumo: todos são equivalentes entre si



2.4 Geração automática de scanners

FLEX

versão mais nova do lex

tradutor de expressão regular em DFA

lê arquivo flex (*arq.l*) e gera código C++ (*lex.yy.c*)

para executar no GNU/Linux:

```
$ flex -fiv arq.l
```

```
$ cc -lf lex.yy.c
```

```
$ a.out
```

Formato de um programa flex (ex. 3.16 Dragão)

```
%{
    Declarações do C (ignorado pelo flex)

}%

Declarações do flex (e.r.)

%%

Expressões regulares : ações ;

%%

Código em C do(a) usuário(a)
```

Exercícios:

3.6 a 3.12, 3.14, 3.16 e 3.17



Matching rule

- De cima para baixo
- Mais longo possível

Elementos sintáticos das e.r. em flex

- [a-zA-Z] conjunto
- a* 0 ou +
- a+ 1 ou +
- b? 0 ou 1
- ^ início da linha
- \$ fim da linha
- [^abc] qualquer caractere exceto a, b, e c.
- {nome} expansão de macro
- . qualquer caractere exceto \n

1. Introdução à Compilação

2. Análise Léxica

3. Análise Sintática

4. Análise Semântica

5. Geração de Código

6. Otimização de Código

3.1 Gramáticas

- Conjunto de regras que descreve as construções válidas de uma linguagem
- Vantagens:
 - Facilita a tradução
 - É precisa
 - É automática
 - Tira ambigüidade
 - Permite tratamento de erro
 - Permite evolução (adaptabilidade)
 - Permite novas construções

3.1.1 Definição

$G = \langle \Sigma, T, NT, S, P \rangle$ onde:

Σ - alfabeto

T - terminais ($T = (\Sigma \cup \epsilon)^+$)

NT - não-terminais

S - símbolo inicial, $S \subseteq NT$

P - produções (regras) da forma $\alpha \rightarrow \beta$



3.1.2 Notação

- $\alpha \rightarrow \beta$ regra
- $\alpha \Rightarrow \beta$ derivação (com a aplicação de uma regra reescreve-se α como β)
- $\alpha \Rightarrow^+ \beta$ derivação (a aplicação de pelo menos uma regra reescreve-se α como β)
- $\alpha \Rightarrow^* \beta$ derivação (a aplicação de zero ou mais regras reescreve-se α como β)
- $\text{exp} \in L(G)$ se e somente se $S \Rightarrow^* \text{exp}$

3.1.3 Classificação de Noam Chomsky

Regras: $\alpha \rightarrow \beta$

Tipo 0 - irrestrita

Nenhuma restrição c.r.a α e β

Tipo 1 - sensível ao contexto

Restrição: $|\alpha| \leq |\beta|$

Tipo 2 - livre de contexto

Restrições: $\alpha \in NT$, $\beta \in (T \cup NT)^+$, $|\alpha| = 1$ e $|\alpha| \leq |\beta|$

Tipo 3 - regular

Restrições: $\alpha, \beta \in NT$

ou $\alpha \in NT$ e $\beta \in T$

ou $\alpha \in NT$ e $\beta = ab$ onde $a \in T$ e $b \in NT$.

3.1.4 Gramáticas regulares

Restrições:

- $\alpha, \beta \in NT$
- ou $\alpha \in NT$ e $\beta = ab$ onde $a \in T$ e $b \in NT$.
- ou $\alpha \in NT$ e $\beta \in T$

Exemplo:

$S \rightarrow aA$

$S \rightarrow bA$

$A \rightarrow b$

➤ Leia Dragão seção 4.3

3.1.4.1 Conversão

De LR para GR:

- strings de a e b começados por a

$S \rightarrow aA$

$A \rightarrow aA$

$A \rightarrow bA$

$A \rightarrow \epsilon$

De er para GR:

1. Converta a e.r. para NFA

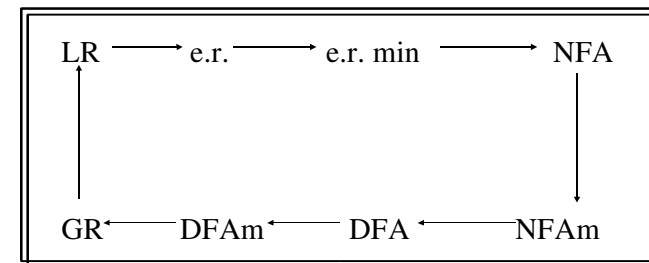
2. Converta o NFA para DFA

3. Para cada transição do DFA $\delta(A,a) = B$, crie a regra

$A \rightarrow aB$



3.1.4.2 Equivalência



3.1.5 Gramáticas livres de contexto

Forma das produções: $\alpha \rightarrow \beta$, onde

- $\alpha \in NT$
- $\beta \in (T \cup NT)^+$
- $|\alpha| = 1$
- $e |\alpha| \leq |\beta|$

– Exemplos: $L = \{a^n v^n \mid 1 \leq n\}$

$S \rightarrow aSb$

$S \rightarrow ab$

Experimente escrever uma GR para L

3.1.5.1 Problemas com GLCs

a) Ambigüidade

Exemplo: gramática para reconhecer expressões aritméticas

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow F$

$F \rightarrow INT$

$F \rightarrow ID$

b) Recursividade à esquerda

Exemplo:

$$E \rightarrow E + E$$

Eliminação:

1. Mude as produções da forma:

$$\begin{array}{l} A \rightarrow A \alpha \\ | \beta \end{array}$$

para:

$$\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \\ | \epsilon \end{array}$$



c) Fatoração

Exemplo:

$$\begin{array}{l} S \rightarrow abcdE \\ | abc \end{array}$$

Eliminação:

1. Mude as produções da forma:

$$\begin{array}{l} A \rightarrow \alpha \beta \\ | \alpha \gamma \end{array}$$

para:

$$\begin{array}{l} A \rightarrow P \beta \\ | P \gamma \\ P \rightarrow \alpha \end{array}$$

d) Não apropriada para descrever a semântica de LPs

Exemplo:

Não existe GLC capaz de reconhecer a linguagem:

$$L = \{ a^n b^m a^n b^m \mid n, m \geq 1 \}$$

Solução:

Uso de GSCs (muito complexo!)

Escrever código para fazer checagem semântica

3.1.5.2 Aplicações

Definição da sintaxe de LPs

Exemplo: definição de variáveis em Pascal

var**a,b,c : tipo;****d : tipo;**

Gramática LC:

$$D \rightarrow \text{var } E$$

$$E \rightarrow F \dots$$

3.1.5.3 Backus-Naur Form

Definição da sintaxe de linguagens artificiais

Exemplo: LPs, *shell* de SOs

Variantes:

Diagramas sintáticos



3.1.6 Árvore de derivação

Def.: Estrutura arborecente resultante de uma derivação

Exemplo: definição de variáveis em Pascal

```
D
  var
    E
      F
      ...
```

3.2 Análise Sintática Descendente

Top Down parsing

Seção 2.4 e 4.4 do Dragão

Tipos

- Análise Sintática Descendente Recursiva
- Análise Sintática Preditiva por Tabela LL

3.2.1 A. S. Descendente Recursiva

A cada regra corresponde um procedimento

Exemplo: Declaração de Variáveis à la Pascal

```
D → var E
E → V : T
V → id
T → int
```

Vantagem: Rápida prototipagem de compiladores

Desvantagem: Lento

Programa equivalente:

```
t: token;
main()
{ t = yylex();
  D();
}
proc D;
{ if t = "var" then t = yylex(); E();
  else erro();
}
proc E;
{ V();
  if t = ":" then t = yylex(); T();
  else erro();
}
```

Riv Rios DC/UFC

04/09/99

45



3.2.2 First

O *first* de um NT é o conjunto de terminais que primeiro podem aparecer em seu lugar durante uma derivação quando ele é reescrito

Algoritmo: FIRST(A)

1. Se há regra $A \rightarrow b\alpha$, ponha b no FIRST(A)
2. Se há regra $A \rightarrow \epsilon$, ponha ϵ no FIRST(A)
3. Se há regra $A \rightarrow BCD$,
 Se $\epsilon \in \text{FIRST}(B)$, ponha FIRST(C) no FIRST(A)
 Senão, ponha FIRST(B) no FIRST(A)

Riv Rios DC/UFC

04/09/99

46

3.2.2 Follow

O *follow* de um NT é o conjunto de terminais que podem vir imediatamente após o NT durante uma derivação quando ele é reescrito

Algoritmo: FOLLOW

1. Ponha $\$$ no FOLLOW(S)
2. Se há regra $A \rightarrow \alpha Ba$, ponha a no FOLLOW(B)
3. Se há regra $A \rightarrow \alpha B\beta$,
 Se $\beta \Rightarrow^* \epsilon$, ponha FOLLOW(A) no FOLLOW(B)
 Senão, ponha FIRST(β) no FOLLOW(B)

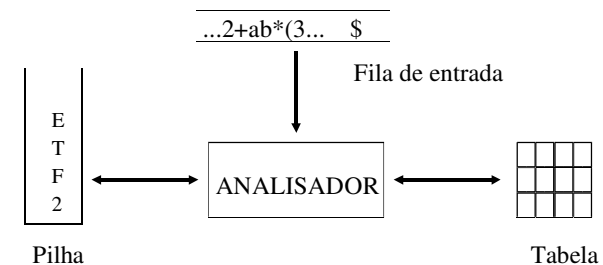
Riv Rios DC/UFC

04/09/99

47

3.2.3 Analisador Sintático Preditivo

O analisador escolhe a regra correta a partir da entrada e do elemento no topo da pilha



Riv Rios DC/UFC

04/09/99

48

3.2.3.1 Construção da Tabela

Algoritmo 4.14 do Dragão:

1. Para cada regra $A \rightarrow \alpha$

Para cada $a \in \text{FIRST}(\alpha)$,

insira $A \rightarrow \alpha$ na posição $[A,a]$ da tabela

Se $\epsilon \in \text{FIRST}(\alpha)$,

insira $A \rightarrow \alpha$ na posição $[A,b]$ da tabela,
onde $b \in \text{FOLLOW}(A)$



3.2.3.2 Gramáticas LL(1)

- Nem toda gramática permite a construção de analisador preditivo por tabela.
- Basta haver duas produções com o mesmo elemento no FIRST
- Isto gera um *conflito*

– Definição:

Uma gramática é LL(1) se, e somente se,
a tabela não tem conflitos

3.2.3.3 Resolução de Conflitos

Problema do *dangling else*

– Considere a gramática:

$S \rightarrow \text{if } E \text{ then } S S'$

| a

$S' \rightarrow \text{else } S$

| ϵ

$E \rightarrow b$

Está gramática não é LL(1).

Qual decisão as LPs tradicionais tomam
para resolver o conflito?

3.2.3.4 Tratamento de Erros

Modo de Pânico - esperar por token de sincronização

Modos de sincronização

(Se não-terminal N está no topo da pilha)

- a) Ignorar o token
- b) Retirar N e esperar por seu *follow* (mais usado!)
- c) Esperar até token do $\text{FIRST}(N)$
- d) Esperar pelo próximo comando
- e) Fazer $N \rightarrow \epsilon$ (se houver tal regra)

(Se terminal a está no topo da pilha)

- a) Desempilhar a e
- b) Imprimir “*a inserido*”

3.3 Gerador de Analisadores Sintáticos

BISON

- versão mais nova do *yacc*
- gera analisador sintático LALR
- lê arquivo com a gramática (prog.y) e código gerado pelo flex (lex.yy.c) e gera código C++ (arq.tab.c)
- para executar no GNU/Linux:

```
$ flex -fiv arq.l
$ bison -v arq.y
$ gcc arq.tab.c
$ a.out
```



Formato de um programa bison

```
%{
    Declarações do C (ignorado pelo flex)
}%
%%
    Declarações do bison (tokens)
%%
    Regras da gramática {ações semânticas} ;
%%
    Código em C do(a) usuário(a)

main() {
    yyparse();          // chamada obrigatória
}
yerror(s)
char *s; {
    fprintf(stderr,"erro: %s na linha %d perto de %s\n",
        s, yylineno, yytext);
}
```

3.3.1 Exemplo de programa Bison

O programa reconhece atribuição de expressões aritméticas simples em notação pós-fixa.

Somente as operações de adição e multiplicação são aceitas.

Exemplo:

x := 2 3 +

ab20c := x 10 5 * ab20c + +

3.3.1.1 Programa flex

```
%option yylineno
ws      [ \t\n]
letter  [a-zA-Z]
digit   [0-9]
%%
{ws}* ;
{letter}({letter}|{digit})* return VARIABLE;
{digit}* return NUMBER;
"+" return PLUS;
"*" return TIMES;
":" return COLON;
":=" return COLON_EQUAL;
. printf("Ligne %d: caractere desconhecido:
  \"%s\" \"\n\",yylineno, yytext);
%%
yywrap() {
    return 1;
}
```

3.3.1.2 Programa Bison

```
%{
    #include <stdio.h>
%}
%token COLON
%token COLON_EQUAL
%token NUMBER
%token PLUS
%token TIMES
%token VARIABLE
%%
assignment : VARIABLE COLON_EQUAL list
            | VARIABLE COLON
              {yyerror("'=' expected");}
            | VARIABLE NUMBER
              {yyerror("'=' expected");}
            | VARIABLE VARIABLE
              {yyerror("invalid id");}
            ;
```

Riv Rios DC/UFC

04/09/99

57



```
list      : item
          | list list PLUS
          | list list TIMES ;
item      : VARIABLE
          | NUMBER ;
%%

#include "lex.yy.c"
yyerror(s)
char *s; {
    if (!strcmp(s,"parse error"))
        printf("Ligne %d: Erro sintatico perto de \"%s\\n\"",
               yylineno,yytext);
    else
        printf("Ligne %d: %s\\n", yylineno, s);
}
main() {
    printf("Digite uma atribuicao: ");
    yyparse(); // Inicio da analise sintatica
}
```

Riv Rios DC/UFC

04/09/99

58

3.4 Análise Sintática Ascendente

(Seção 4.5 do Dragão)

Uso de handles

S → **a A D e**

A → **a A**

| b

D → **f**

Ao reconhecer **aA** na entrada, o analisador terá encontrado o lado direito da segunda regra e poderá, então, substituí-lo pelo não-terminal **A**, como se **aA** fosse um *handle* usado para se subir na árvore sintática correspondente à sequência de entrada.

Riv Rios DC/UFC

04/09/99

59

3.4.1 Análise LR

Método para análise ascendente

LR(1) - *left-to-right, rightmost, lookahead of 1*

Operações:

- *shift* - empilhar
- *reduce* - reduzir, usando uma regra
- aceitar
- rejeitar - erro

Estruturas de dados:

- pilha
- fila de entrada

Riv Rios DC/UFC

04/09/99

60

3.4.2 Conflitos

a) *shift/reduce*

Entrada na tabela contém empilhamento e redução

```
S → if E then S
      | if E then S else S
      | a
E → b
```

Entrada: **if b then a else a**

Ações:

- empilha *else*
- reduz pela regra 1



b) *reduce/reduce*

Redução por mais de uma regra é possível

```
defproc → PROC ID ( listapar )
listapar → ID
           | listapar , ID
defvar → listavar : tipo
listavar → ID
           | listavar , ID
```

Entrada: **proc p (a, b)**

Ações:

- reduz por **listapar**
- reduz por **listavar**

3.4.3 Gramáticas Ascendentes

a) SLR - simple LR

Tabela é gerada a partir dos estados canônicos

Redução é feita pelo FOLLOW

Pode gerar conflitos em alguns casos

b) LR

Redução é feita por subconjunto do FOLLOW

c) LALR

Tabela é gerada a partir dos estados canônicos da tabela LR,

juntando-se estados com lados esquerdos iguais

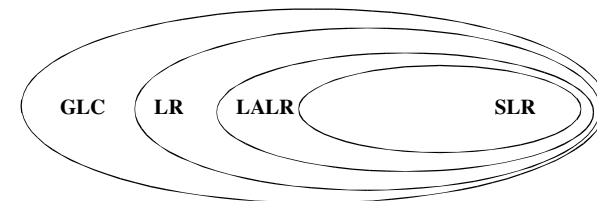
Redução é feita por subconjunto do FOLLOW

Número de estados é o mesmo da SLR

Erros são reconhecidos mais cedo que na SLR

Resumo:

- Gramáticas SLR são mais facilmente geradas, mas podem gerar conflitos
- Gramáticas LALR têm o mesmo tamanho, não geram conflitos, descobrem erro mais cedo e são usadas pelo Bison
- Gramáticas LR são mais gerais, maiores e mais lentas



3.5 Construção da Árvore Sintática no Bison

Estrutura de um nó da árvore

- Rótulo (Tipo)
- Texto ou número (*union* do C)
- Ponteiro para operando 1
- Ponteiro para operando 2
- Ponteiro para operando 3
- Ponteiro para próximo



3.5.1 Tabela de Rótulos

Folhas

- T_INT T_ID

Comandos

- T_ASGN T_IF T_WHIL

Operadores aritméticos

- T_PLUS T_SUBT T_MULT T_DIVD

Operadores lógicos

- T_AND T_OR T_NOT

Operadores relacionais

- T_EQ T_LT T_LE T_GT T_GE T_NEQ

Definições

- T_VARD T_CNTD T_EQD T_PROC

Outros

- T_PRGM T_BLK

1. Introdução à Compilação
2. Análise Léxica
3. Análise Sintática
4. Análise Semântica
5. Geração de Código
6. Otimização de Código

4.1 Análise Semântica

Função: Checagem de tipos

- Estática
- Dinâmica
- Coerção

Detalhes

- Unicidade de identificadores
- Definição
- Listas de parâmetros e índices
- Fluxo de controle de execução

Ações semânticas de BISON podem ser usadas

4.2 Expressão de Tipos

- Simples: int, real, char, VOID
- Vetor:
 - **Array**[-1..10] **of** real;
 - array(-1..1, real)
- Registro:
 - **Record** x:int; y:real **end**
 - record(x X int, y X real)
- Função:
 - **Function** f(i,j:int) : real;
 - int X int \rightarrow real
- Ponteiro:
 - **Pint** = ^int;
 - pointer(int)



4.3 Equivalência de Tipos

- Estrutural - C
- Nominal- Pascal

Algoritmo para EN:

```
Function Equiv (T1,T2) : Boolean;
If T1 in [int,real,bool,char,void] and
    T2 in [int,real,bool,char,void] and
    T1 = T2
Then return true
Elsif T1=array(I1,S1) and T2=array(I2,S2)
Then return Equiv(I1,I2) and Equiv(S1,S2)
```

```
Elsif T1=pointer(S1) and T2=pointer(S2)
Then return Equiv(S1,S2)
Elsif T1=record(L1) and T2=record(L2)
Then return Equiv(L1,L2)
Elsif T1=L1  $\rightarrow$  S1 and T2=L2  $\rightarrow$  S2
Then return Equiv(L1,L2) and Equiv(S1,S2)
Elsif T1=V1xS1 and T2=V2xS2
Then return Equiv(V1,V2) and Equiv(S1,S2)
Else return false;
End Function;
```

4.4 Tabela de Símbolos

- Def.: Tabela com todos os símbolos do programa e informações sobre eles

- Árvore de símbolos seria mais apropriado

Estrutura de um nó da árvore

- Nome
- Ponteiro para descendente
- Ponteiro para ancestral imediato
- Ponteiro para lista de identificadores locais com
 - nome, tipo, próximo
- Ponteiro para irmãos

Exercícios:

4.1 a 4.6, 4.8 a 4.10 e 4.13 a 4.15

1. Introdução à Compilação
2. Análise Léxica
3. Análise Sintática
4. Análise Semântica
5. Geração de Código
6. Otimização de Código



5.1 Etapas

- Geração de código intermediário
- Otimização de código intermediário
- Geração de código objeto
- Otimização de código objeto

Vantagens de usar CI:

- portabilidade
- construção de interpretador independente de máquina
- otimização de código independente de máquina
- geração de código para diferentes plataformas a partir de um único compilador

5.2 Geração de código intermediário

- Tradução dirigida pela sintaxe
 - geração feita a medida que o código vai sendo analisado
 - corresponde às ações semânticas do Bison
- Exemplo para máquina com:
 - Código de três endereços
 - pilha
- Código gerado:
 - a nível de expressão
 - a nível de comando

5.3 Linguagem para Código Intermediário

- Comandos do tipo
 - `x := y`
 - `x := y op z`
 - `goto L`
 - `if x oprel y goto L`
 - `x := op y`

1. Introdução à Compilação
2. Análise Léxica
3. Análise Sintática
4. Análise Semântica
5. Geração de Código
6. Otimização de Código



6.1 Técnicas

a) Eliminação de subexpressões comuns

t6 := 4*i		t6 := 4*i
x := a+0		x := a+0
t7 := 4*i		t7 := t6 ✓
t8 := 4*j		t8 := 4*j
t9 := a	→	t9 := a
a := t7+t8		a := t7+t8
t10 := 4*j		t10 := t8 ✓
b := x+a		b := x+a
jump L1		jump L1

b) Propagação de cópia

t6 := 4*i		t6 := 4*i
x := a+0		x := a+0 ✓
t7 := t6		
t8 := 4*j		t8 := 4*j
t9 := a	→	t9 := a ✓
a := t7+t8		a := t6+t8
t10 := t8		t10 := t8
b := x+a		b := x+a
jump L1		jump L1

c) Eliminação de código inútil

t6 := 4*i		t6 := 4*i
x := a+0		x := a+0
t8 := 4*j		t8 := 4*j ✓
t9 := a	→	
a := t6+t8		a := t6+t8 ✓
t10 := t8		
b := x+a		b := x+a
jump L1		jump L1

d) Identidades algébricas

t6 := 4*i	t6 := 4*i
x := a+1	x := a ↙
t8 := 4*j	t8 := 4*j
t9 := a	
a := t6+t8	a := t6+t8
t10 := t8	
b := x+a	b := x+a
jump L1	jump L1



Resultado final

t6 := 4*i	t6 := 4*i
x := a	x := a
t8 := 4*j	t8 := 4*j
a := t6+t8	a := t6+t8
b := x+a	b := x+a
jump L1	jump L1

F I M