Contents

1 Interpretação Pura

Uma interpretação pura em compiladores é feita da seguinte maneira: dado o programa fonte, o programa fonte passará por um interpretador. Esse **interpretador** irá interpretar o código fonte a partir de dados de entrada e bibliotecas, gerando assim os resuldados.

1.1 Benefícios

- Os erros são mais fácies de serem detectados
- Além de testar pedaços de códigos

1.2 Desvantagens

- A execução é mais lenta

2 Compilação para bytecode

O uso de compilação para geração de bytecode faz com que o programa fonte passe pelas seguintes partes:

2.1 Analisador léxico

O analisador léxico verifica se os elementos presentes no código fonte são válidos na linguagem. O seu papel é segmentar o programa e verificar a validade de cada segmento de código. A saída do analisador léxico são tokens. Tokens possuem um lexema, que nada mais é do que a classificação do token, linha, coluna e valor do token, que nada mais é do que o conteúdo em si.

2.2 Analisador sintático

Tem como saída uma árvore de derivações. Esse analisador tem como objetivo requisitar tokens do analisador léxico e a partir desses tokens fazer uma análise sintática do programa fonte. A medida que a análise sintática é feita, o analisador vai montando a árvore de derivação. Existem vários tipos de analisadores sintáticos.

2.3 Analisador semântico

A partir da ávore de derivação criada pelo analisador sintático, o analisador semântico avalia a compatibilidade dos nós e folhas dessa árvore. Ou seja, é no analisador semântico que a verificação dos tipos é efetuada, vendo assim se o programa fonte faz sentido segundo as especificações da linguagem.

2.4 Gerador de bytecode

Esse processo não é coberto pela disciplina.

2.5 Tabela de símbolos

Estabelece uma comunicação entre fases(citadas acima) diferentes do compilador. Geralmente na tabela de símbolos temos informações a respeito de variávies, funções/procedimentos e parâmetros.

3 Análise Léxica(Scanner)

Lê o código fonte e o separa em lexemas(unidades léxicas). Identifica os lexemas, categorizando-os e guardando suas posições. Elimina comentários.

Lexemas serão palavras reservadas ou não. De modo geral os lexemas podem ser :

- Identificadores
- Constantes
- Operadores
- Separadores
- Palavras reservadas

Para fazer a identificação de cada um desses tipos, pode-se usar:

- Expressões regulares
- Gramática regular
- Autômato finito determinístico
- Autômato finito não-determinístico

O alfabeto léxico se trata dos caracteres permitidos no código fonte, enquanto o alfabeto sintático se refere aos tokens possíveis.

4 Análise sintática(Parser)

É a segunda fase de um compilador. Atua sobre uma GLC, dado o programa fonte e seus respectivos tokens, produz-se uma árvore de derivação que reflete a estrutura sintática da sentença.

Existem algumas estratégias de análise:

- Top-down ou descendente
  - Backtracking
  - Preditiva recursiva
  - Preditiva tabular
- Bottom-up ou ascendente(redutiva, empilha-reduz)
  - Precedência de operadores
  - SLR(Simple left to right rightmost)

5 Análise Top-Down

A análise se inicia pela raiz da árvore através de derivação. Produções ao invés de erivações são escolhidas com base na entrada.

5.1 Análise com Backtracking

A cada derivação com mais de uma opção, cria-se um checkpoint que permite retorno em caso de erro

5.1.1 Desvantagens

- É difícil encontrar a causa do erro
- Ineficiente

5.1.2 Vantagens

- É capaz de analisar qualquer tipo de gramática tratável

5.2 Análise recursiva preditiva

A gramática não pode ter recursões à esquerda. Logo ela deve ser:

- Fatorada
- Primeiros terminais deriváveis de produções de um mesmo não-terminal não podem ter intersecções

5.2.1 Características

- Fácil de implementar
- Muito código
- Permite tratar repetições e opcionalidades

Nesse tipo de analisador, o código é tratado sintáticamente da esquerda para a direita, onde para cada símbolo não terminal existe uma função que trata esse símbolo.

5.3 Análise preditiva tabular(não recursiva)

Para esse tipo de análise a gramática também deve ser fatorada e os primeiros temrinais deriváveis de um mesmo não-terminal não podem ter intersecções.

5.3.1 Funcionamento

1. Inicia-se o processamento lendo o primeiro token e com o símbolo inicial na pilha.
2. Se entrada for EOF e a pilha estiver vazia a sentença é aceita.
3. Se token na entrada corresponder ao terminal no topo da pilha, o token foi reconhecido, acessa-se o próximo token e desempilha-se o terminal
4. Topo da pilha não-terminal, indexa-se a tabela de análise pelo mesmo e pelo token na entrada, se a casa indexada estiver vazia é um erro sintático, caso contrário, efetua-se a derivação indicada.
   (a) Desempilha-se o não-terminal e empilha o lado direito da produção na ordem inversa.

5.3.2 Exemplo:

- Gramática
  (1) Eb = Tb Ebr
  (2) Ebr = 'ou' Tb Ebr
  (3) Ebr = Vazio
  (4) Tb = Fb Tbr
  (5) Tbr = 'e' Fb Tbr
  (6) Tbr = Vazio
  (7) Fb = 'não' Fb
  (8) Fb = '(' Eb ')'
  (9) Fb = 'id'
  (10) Fb = 'verd'
  (11) Fb = 'falso'

- Tabela de análise com recuperação de erro local

  Apenas uma modificação é feita:

  - Desempilha um símbolo
  - Empilha um símbolo
  - Introduz um terminal na entrada
  - Trocar terminal da entrada

  Tabela:

  - é adicionada uma linha para os terminais que apareçam fora da 1° posição em algum handle.
  - Expande as produções vazias para posições em branco na mesma linha (porterga a identificação de erro)

  Tipos de erros encontrados no exemplo:

  - Err1:
    * Msg: "Operando esperado"
    * Ação: "Insere "id" na entrada"
  - Err2:

    - Msg: " ')' esperado"
    - Ação: Desempilha ')'
  - Err3:
    - Msg: "EOF esperado"
    - Ação: Elimina a lista restante da entrada

	'ou'	'e'	'nao'	'('	')'	'id'	'verd'	'falso'	EOF
Eb	Err1	Err2	d1	d1	Err1	d1	d1	d1	Err1
Ebr	d2	Vazio	Vazio	Vazio	Vazio-d3	Vazio	vazio	Vazaio	Vazio-d3
Tb	Err1	Err1	d4	d4	Err1	d4	d4	d4	Err1
Tbr	Vazio-d6	d5	Vazio	Vazio	Vazio-d6	Vazio	Vazio	Vazio	Vazio-d6
Fb	Err1	Err1	d7	d8	Err1	d9	d10	d11	Err1
')'	Err2	Err2	Err2	Err2	DAT	Err2	Err2	Err2	Err2
PV	Err3	Err3	Err3	Err3	Err3	Err3	Err3	Err3	AC

- Tabela de análise com recuperação de erro pânico
  É necessário identificar os tokens de sincronização. Para cada não terminal temos que o token sync desse não terminal é dado pelo follow dele.

  Tratamento:

  - Mensagem de erro
  - EOF na entrada: encerra a análise
  - Pilha:
    - Terminal:
      - Despreza tokens na entrada enquanto terminal da pilha for diferente do token de entrada
      - Se terminal corresponder ao token de entrada continua a análise
    - Não-terminal
      - Despreza tokens na entrada enquanto entrada não corresponder a um terminal de sincronização
      - Quando corresponder: desempilha o não terminal e continua a análise

	'ou'	'e'	'nao'	'('	')'	'id'	'verd'	'falso'	EOF
Eb	-	-	d1	d1	Sync	d1	d1	d1	Sync
Ebr	d2	-	-	-	Vazio-d3	-	-	-	Vazio-d3
Tb	-	-	d4	d4	Sync	d4	d4	d4	Sync
Tbr	Vazio-d6	d5	-	-	Vazio-d6	-	-	-	Vazio-d6
Fb	Sync	Sync	d7	d8	Sync	d9	d10	d11	Sync

6 Análise bottom-up

- Redutiva
- Empilha/reduz
- A partir das folhas da árvore
- Inicia com a pilha vazia
- Empilha símbolos até que na pilha estejam símbolos que correspondam ao lado direito de uma produção (**handle**)
- Este **handle** é desempilhado e em seu lugar é empilhado o símbolo do lado esquerdo da produção(redução)
- São empilhados tokens ou símbolos não terminais quando ocorre uma redução
- A produção a ser usada só é conhecida no momento da redução
- Aceita quando na pilher tiver o símbolo inicial da GLC e na entrada EOF
- Se a gramática for não ambigua, então toda forma sentencial gerada por G tem exatamente um handle.

6.1 Analisadores

- Ações:
  - Empilha (EAT): empilha o token na entrada e acessa o próximo token
  - Reduz(ri): substitui o handle da produção (i) pelo terminal à esquerda nesta produção

  - Aceita(Ac): reconhece a sentença quando na pilha tem o símbolo inicial e EOF na entrada
  - Erro: demais casos

- Tipos de analizadores:
  - Precedência de operadores: bom para reconhecimento de expressões
  - LR:
    * SLR: Simple LR (só veremos esse)
    * LALR: Look Ahead LR
    * LR canônico

6.2 Analisador de precedência de operadores

- Gramática de operadores
- Gramática simplificada
- Não pode ter dois não terminais adjascentes
- Não podem ter produções vazias
- Não ode ter um operador com aridades diferentes
- A gramática não tem informações de rpecedência e associatividade
  - esta informação deve ser colocada na tabela de análise
- Tabela de análise:
  - Tokens na pilha x tokens na entrada, PST(pilha sem token)
  - entrada(colunas): tokens e EOF
  - PST pode estar vazia (PV) ou não ter um não terminal

6.2.1 Relação entre tokens + ao topo topo/PST e a entrada(tokens ou EOF)

- Prioridade calculada em relação da precedência e da associatividade
- Se token na pilha tem prioridade:
  - maior: reduz
  - menor: empilha entrada

  - igual: empilha entrada

- APO não garante validade do handle
- Na redução os elementos da pilha devem ser testados para garantir esta validade
  - Caso isso não ocorra, temos um erro na redução chamado de erro mais tarde
- Erros identificados durante a análise, especificados na tabela de análise, são ditos erros mais cedo

6.2.2 Exemplo

- Gramática
  (1) Eb = Eb 'ou' Eb
  (2) —- Eb 'e' Eb
  (3) —- 'nao' Eb
  (4) —- '(' Eb ')'
  (5) —- 'id'
  (6) —- 'verd'
  (7) —- 'falso

Tipos de erro mais cedo encontrados:

- Err1: msg: ')' esperado ação: encerra análise
- Err2: msg: operador binário esperado ação: insere um operador binário
- Err3: ')' não esperado ação: remove ')' da entrada

Erros mais tarde que podem ocorrer:

- (1) e (2):
  Se topo != Eb
  msg: Segundo operando faltando
  senão desempilha Eb
  desempilha 'ou'/'e'
  se topo != Eb
  msg: Primeiro operando faltando
  senão desempilha Eb
  empilha novo Eb

- (3):
  se topo != Eb msg: operando faltando senão desempilha Eb desempilha 'nao' empilha novo Eb
- (4):
  desempilha ')' se topo != Eb msg: expressão booleana faltando senão desempilha Eb desempilha '(' empilha novo Eb
- (5), (6) e (7): não existe o erro porque nunca é empilhado nada sobre esses terminais.

	'ou'	'e'	'nao'	'('	')'	'id'	'verd'	'falso'	EOF
'ou'	r1	EAT	EAT	EAT	r1	EAT	EAT	EAT	r1
'e'	r2	r2	EAT	EAT	r2	EAT	EAT	EAT	r2
'nao'	r3	r3	EAT	EAT	r3	EAT	EAT	EAT	r3
'('	EAT	EAT	EAT	EAT	EAT	EAT	EAT	EAT	Err1
')'	r4	r4	Err2	Err2	r4	Err2	Err2	Err2	r4
'id'	r5	r5	Err2	Err2	r5	Err2	Err2	Err2	r5
'verd'	r6	r6	Err2	Err2	r6	Err2	Err2	Err2	r6
'falso'	r7	r7	Err2	Err2	r7	Err2	Err2	Err2	r7
PST	EAT	EAT	EAT	EAT	Err3	EAT	EAT	EAT	AC

6.3 Analisador SLR

Conjunto canônico de itens SLR

- Item Lr(0) em uma GLC marca uma posição no andamendo da análise marcada por um ponto.

6.3.1 Exemplo

- Gramática
  (0) S = Eb
  (1) Eb = Eb 'ou' Tb
  (2) Eb = Tb
  (3) Tb = Tb 'e' Fb
  (4) Tb = Fb
  (5) Fb = 'nao' Fb
  (6) Fb = '(' Eb ')'
  (7) Fb = 'id'

Fazendo o fechamento da gramática

- 0 = {S = . Eb, Eb = . Eb 'ou' Tb, Eb = . Tb, Tb = . Tb 'e' Fb, Tb = . Fb, Fb = . 'nao' Fb, Fb = . '(' Eb ')', Fb = . 'id'}
- 1 = (0, Eb) = {S = Eb ., Eb = Eb . 'ou' Tb}
- 2 = (0, Tb) = {Eb = Tb ., Tb = Tb . 'e' Fb}
- 3 = (0, Fb) = {Tb = Fb .}
- 4 = (0, 'nao') = { Fb = 'nao' . Fb, Fb = . 'nao' Fb, Fb = . '(' Eb ')', Fb = . 'id'}
- 5 = (0, '(') = { Fb = '(' . Eb ')', Eb = . Eb 'ou' Tb, Eb = . Tb, Tb = . Tb 'e' Fb, Tb = . Fb, Fb = . 'nao' Fb, Fb = . '(' Eb ')', Fb = . 'id'}
- 6 = (0, 'id') = { Fb = 'id' .}
- 7 = (1, 'ou') = { Eb = Eb 'ou' . Tb, Tb = . Tb 'e' Fb, Tb = . Fb, Fb = . 'nao' Fb, Fb = . '(' Eb ')', Fb = . 'id'}
- 8 = (2, 'e') = { Tb = Tb 'e' . Fb, Fb = . 'nao' Fb, Fb = . '(' Eb ')', Fb = . 'id'}
- 9 = (4, Fb) = {Fb = 'nao' Fb .}
- (4, 'nao') = 4
- (4, '(') = 5
- (4, 'id') = 6
- 10 = (5, Eb) = {Fb = '(' Eb . ')', Eb = Eb . 'ou' Tb}
- (5, Tb) = 2
- (5, Fb) = 3
- (5, 'nao') = 4
- (5, '(') = 5
- (5, 'id') = 6
- 11 = (7, Tb) = { Eb = Eb 'ou' Tb ., Tb = Tb . 'e' Fb }

- (7, Fb) = 3
- (7, '(') = 5
- (7, 'nao') = 4
- (7, 'id') = 6
- 12 = (8, Fb) = { Tb = Tb 'e' Fb .}
- (8, 'nao') = 4
- (8, '(') = 5
- (8, 'id') = 6
- 13 = (10, ')') = { Fb = '(' Eb ')' .}
- (10, 'ou') = 7
- (11, 'e') = 8