Universidade Federal de Mato Grosso do Sul
Faculdade de Computação
Linguagens Formais e Autômatos - T1
Profa. Edna A. Hoshino

Trabalho Prático - autômatos finitos × analisadores léxicos + autômato de pilha × analisadores sintáticos

1 Analisador léxico e sintático para a linguagem C-

Um compilador é um programa que traduz um código de uma linguagem-fonte para um código em uma linguagem-alvo. Para isso, os compiladores basicamente realizam duas etapas: análise e síntese. Na fase de análise, o código na linguagem-fonte é analisada do ponto de vista léxico, sintático e semântico, enquanto na síntese, um código equivalente é gerado na linguagem-alvo.

O trabalho prático refere-se ao uso de autômatos finitos determinísticos e autômatos com pilha para realizar, respectivamente, a análise léxica e sintática de programas escritos em um subconjunto da linguagem C, o qual denominaremos linguagem C-.

A linguagem C- é aquela gerada pela seguinte gramática $G = (V, T, P, S)$ em que P é constituída pelas seguintes produções:

```
S         -> Function S_
S_        -> Function S_
           | epsilon
Function  -> Type Function_
Type      -> void
           | int
           | float
Function_ -> main() { B }
           | id() { B }
B         -> C B
           | epsilon
C         -> id = E ;
           | while (E) C
           | { B }
E         -> T E_
E_        -> + T E_
           | epsilon
T         -> F T_
T_        -> * F T_
           | epsilon
F         -> ( E )
           | id
           | num
```

Os conjuntos de variáveis V e de terminais T são dados abaixo:

```
V = {S, S_, Function, Function_, Type, C, B, E, T, E_, T_, F}

T= {ELSE, FLOAT, FOR, IF, INT, MAIN, VOID, WHILE, ID, NUM, OP_ATRIB, OP_ADIT, OP_MULT,
    OP_REL, ABRE_PARENT, FECHA_PARENT,PONTO_VIRG, VIRG, ABRE_CHAVES, FECHA_CHAVES, FIM,
    INVALIDO}
```

1.1 Análise Léxica

Um analisador léxico é o componente de um compilador responsável por reconhecer os terminais da gramática da linguagem-fonte.

O autômato finito determinístico M é aquele que aceita o conjunto das palavras associadas aos terminais T. O autômato M processa os caracteres de um arquivo de entrada, reconhecendo os terminais em T ou identificando um terminal INVALIDO. Os espaços em branco, tabulações e fim de linha são ignorados por M.

Considere o seguinte exemplo de programa escrito em C-:

```
void main()
{
   a 10 =;
   while (a!=0)
   {
     a = @a-1;
}
```

O processamento do autômato M no exemplo acima resultará no reconhecimento da seguinte sequencia de terminais:

```
VOID
MAIN
ABRE_PARENT
FECHA_PARENT
ABRE_CHAVES
ID
NUM
OP_ATRIB
PONTO_VIRG
WHILE
ABRE_PARENT
ID
OP_REL
NUM
FECHA_PARENT
ABRE_CHAVES
ID
OP_ATRIB
INVALIDO
ID
OP_ADD
NUM
PONTO_VIRG
FECHA_CHAVES
```

Note que o analisador léxico não se preocupa com a ordem em que os terminais aparecem no programa nem na quantidade delas (o analisador léxico não verificou que o comando de atribuição está incorreto nem que falta '}'). O analisador léxico verifica somente a ordem em que os caracteres aparecem no arquivo de entrada, ou seja, no programa, para reconhecer um terminal.

O autômato M é implementado através de uma tabela de transições armazenada em uma matriz de dimensão $|Q| \times |\Sigma|$, onde Q é o conjunto de estados de M e Σ o alfabeto dos símbolos de entrada. O alfabeto é armazenado em um vetor de caracteres. No entanto, três símbolos adicionais também são considerados: LETRA, DIGITO e INVALIDO. O primeiro representa qualquer letra, o segundo um dígito qualquer e o último um símbolo não pertencente ao alfabeto. Este último é incluído para permitir ao autômato tratar os casos em que símbolos desconhecidos aparecem na entrada.

1.2 Análise Sintática

Um analisador sintático é o responsável por reconhecer as palavras geradas pela gramática da linguagem-fonte. Note que as palavras geradas por G equivalem aos programas que respeitam as regras (ou produções) da gramática C-.

Para explicar o funcionamento de um analisador sintático, considere o seguinte exemplo de programa:

```
void main()
{
   a = 10;
   b = 3;
   while (a){
     c = (a+b*2;
     a = a - 1;
     b = b*b;
   }
}
```

Para o exemplo acima, um analisador sintático reportaria uma mensagem de erro como apresentado abaixo para indicar que um ")" era esperado e não foi encontrado:

```
Total de linhas processadas: 6
Resultado: Esperado token 15 (FECHA_PARENT)
```

Nesse trabalho, é fornecido o código de um analisador sintático descendente recursivo que simula a execução de um autômato com pilha determinístico M' e aceita a linguagem gerada por uma gramática G.

A construção do analisador sintático descendente recursivo exige que a gramática G seja do tipo LL(1). Gramáticas do tipo **LL(1)** são aquelas que podem gerar todas as palavras de uma linguagem usando apenas 1 símbolo por vez da palavra, começando pelo símbolo mais à esquerda (**L**eft) e fazendo derivações mais à esquerda (**L**eft) de forma determinística.

Formalmente, uma gramática é LL(1) se e somente se, sempre que $A \to \alpha|\beta$ forem duas produções distintas da gramática as seguintes condições devem ser satisfeitas:

- α e β não derivam em palavras começando pelo mesmo terminal;
- ambos não podem derivar ε;
- Se β derivar em ε então α não pode derivar palavras que começam com um terminal em FOLLOW(A).

Observe que gramáticas ambíguas e gramáticas com recursão mais à esquerda não são LL(1).
Para entender melhor o conceito de gramática LL(1), algumas definições serão necessárias:

- $FIRST(A) = \{x \in T : A \Rightarrow^* x\alpha$, com $\alpha \in (T \cup V)^*\}$, ou seja, $FIRST(A)$ é o conjunto dos terminais que aparecem no início das formas sentenciais derivadas a partir de A. Aqui, $A \in (V \cup T)^*$. Note que $FIRST(A) = \{A\}$, se $A \in T$.
- $FOLLOW(A) = \{x \in T : S \Rightarrow^* \alpha A x \beta$, com $\alpha, \beta \in (T \cup V)^*\}$, ou seja, $FOLLOW(A)$ é o conjunto dos terminais que aparecem imediatamente à frente de A em alguma forma sentencial derivada a partir de S. Aqui, A é uma variável.

Para calcular $FIRST(A)$ e $FOLLOW(A)$ para cada variável A faça:

1. Para cada produção $A \to B_1 B_2 \dots B_k$, coloque a em $FIRST(A)$ se $a \in FIRST(B_i)$, para algum i, e $B_1, B_2, \dots, B_{i-1}$ forem anuláveis. Em outras palavras, adicionamos a $FIRST(A)$ todo terminal a que está em $FIRST(B_1)$ e, se B_1 for anulável, colocamos também todo terminal a que está em $FIRST(B_2)$ em $FIRST(A)$ e assim sucessivamente.

2. Coloque o terminal FIM, que é um indicador de fim da palavra de entrada (no caso de um programa, seria o marcador de final de arquivo) em $FOLLOW(S)$ para a variável inicial S da gramática.

3. Coloque $FIRST(B_1)$ em $FOLLOW(A)$ se houver uma produção $X \to \alpha A B_1 B_2 \dots B_k$ e, caso B_1 seja anulável, coloque também todo terminal que está em $FIRST(B_2)$ em $FOLLOW(A)$ e, assim sucessivamente. Se $B_1, B_2, \dots, B_k$ forem todos anuláveis, coloque todo terminal que está em $FOLLOW(X)$ em $FOLLOW(A)$.

Por exemplo,

```
FIRST(S) = FIRST(Function) = FIRST(Type) = {int, float, void}
FIRST(E) = FIRST(T) = {ABRE_PARENT, NUM, ID}
FOLLOW(F) = {FECHA_PARENT, OP_ADIT, OP_MULT, PONTO_VIRG}
FOLLOW(S_) = FOLLOW(S) = {FIM}
```

Seja A uma variável da gramática G e $A \to \alpha_1 | \alpha_2 | \dots | \alpha_k$ as produções de A. A partir da definição de $FIRST()$ e $FOLLOW()$, pode-se caracterizar as gramáticas LL(1) sendo aquelas em que:

(1) $FIRST(A) \cap FOLLOW(A) = \emptyset$, sempre que A for anulável, ou seja $A \Rightarrow^* \varepsilon$;

(2) $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$, para todo $i \neq j$, $1 \leq i, j \leq k$.

Nas situações em que a última condição não é satisfeita pela gramática, pode-se aplicar a fatoração nas produções até que a condição seja satisfeita. Por exemplo, se

$$A \to id + A | id - A$$

basta trocar essas duas produções por

$$A \to id A'$$

e

$$A' \to +A | -A$$

Dado que G satisfaz as condições acima, o analisador sintático descendente recursivo é obtido associando-se uma função para cada variável. Essa função simula as derivações com respeito às produções da variável. Uma vez que a condição (1) e (2) são respeitadas pela gramática, é possível decidir qual produção usar a cada passo de derivação olhando-se apenas o próximo símbolo da palavra (denominado `lookahead`).

Seja A uma variável e sejam as seguintes produções de A:

$$A \to aA | bbB | c$$

A função associada à variável A é

```
void A()
{
  if (lookahead==a){ /* usa a producao A-> aA */
      match(a);
      A();
  }
  else if (lookahead==b){   /* usa a producao A-> bbB */
      match(b);
      match(b);
      B();
  }
  else{  /* usa a producao A-> c */
      match(c);
  }
}
```

Note que a decisão por qual produção derivar é tomada com base no `lookahead`. A função `match(t)` verifica se o próximo símbolo é igual ao símbolo esperado `t` e avança o ponteiro da fita de entrada para atualizar o lookahead. Note que em caso de erro, a função mostra uma mensagem de erro e aborta o programa.

```
void match(int t)
{
  if (lookahead==t){
      lookahead = lex();
  }
  else{
      printf("\nErro: simbolo %d esperado.", t);
      exit(1);
  }
}
```

Apenas para complementar, suponha que a variável B tenha uma produção vazia, por exemplo, $B \rightarrow bBb|\varepsilon$. A função associada a B seria:

```
void B()
{
  if (lookahead==b){ /* usa a producao B-> bBb */
      match(b);
      B();
      match(b);
  }
}
```

Note que se o lookahead for diferente de b, a função B não faz nada, o que equivale a realizar a derivação com a produção vazia $B \rightarrow \varepsilon$.

1.3 Objetivo

O objetivo deste trabalho é alterar o analisador sintático para aumentar o conjunto de palavras geradas pela gramática da linguagem C-.

Cada grupo (de exatamente três acadêmicos) deve selecionar uma das propriedades listadas abaixo e que ainda estão inativas no compilador da linguagem C- e, então, deve alterar os programas fornecidos pelo professor para que o analisador léxico e sintático reconheça programas sintaticamente corretos satisfazendo a propriedade escolhida:

1. uso de rótulos (`labels`), uso de desvio incondicional (`goto`) e expressão com operador condicional ternário (por exemplo, $a > b?c : d$);

2. uso de especificadores de tipo (`static, extern, typedef`) e da estrutura `union`;

3. expressões aritméticas envolvendo operadores binários ($-$ e $/$), operadores de incremento e decremento ($++$ e $--$), operadores de atribuição ($+=$, $-=$, $*=$, $/=$), sendo que algumas dessas alterações devem ser feitas no léxico, e declaração de variáveis globais do tipo básico;

4. especificação de função com ou sem parâmetros e chamada de função com ou sem passagem de argumentos (na forma de um comando ou como termo de uma expressão);

5. ignorar comentários de linha e de bloco (esta alteração deve ser feita no analisador léxico e não no sintático) e declaração de estruturas (`struct`);

6. expressões lógicas, relacionais e comando `for`;

7. analisador léxico implementado sem o uso de tabela de transições (isto é, com a implementação das transições do autômato diretamente no código usando-se `switch-case`);

8. declaração e uso de vetores de uma ou várias dimensões com ou sem inicialização de valores;

9. especificação de enumerações (`enum`) e declaração e uso de variáveis do tipo ponteiros (inclusive de ponteiro para ponteiro);

10. alguma funcionalidade da linguagem Python que não existe na linguagem C:

    (a) especificação e uso de dicionário (`dictionary`);

    (b) construção de listas (`list`) e uso de abrangência de listas (`list comprehensions`);

    (c) construção e uso de strings, bem como de fatias da string (`slices`);

    (d) alguma outra funcionalidade, desde que em concordância do professor.

Para realizar estas atividades, cada grupo potencialmente incluirá novas variáveis na gramática G e novas produções. Outras alterações pontuais, como aumentar o valor de algumas constantes podem ser necessárias.

1.4 Observações importantes

Para entregar o seu trabalho corretamente, observe os itens listados abaixo:

- o trabalho deve ser feito por grupos de **três a quatro alunos** e **deve ter a participação de todos os integrantes do grupo na sua execução**. Se necessário, o professor pode **realizar arguições individuais e/ou em grupo** para verificar se este requisito foi cumprido. **A não participação efetiva de algum integrante implicará na nota ZERO ao trabalho do grupo!**

- qualquer forma de plágio não será aceita. Se identificado plágio, a nota de cada grupo envolvido será ZERO!

- a programação deve ser feita em linguagem `C` e compilável em uma instalação Linux padrão ANSI (com `gcc`.

- um relatório de no máximo 3 páginas (limite rígido) deve ser entregue junto com o código, para detalhar as alterações realizadas na gramática G. Descreva no relatório, as novas variáveis e produções adicionadas na gramática antes e depois de torná-la LL(1).

- o trabalho deve ser entregue em um arquivo formato `tgz` enviado via *EAD* até as 23:59 horas da data fixada para a entrega na página da disciplina. Ao ser descompactado, este arquivo deve conter o diretório `Nome1-Nome2-Nome3` contendo os subdiretórios `Programas` e `Texto`, onde Nome1, Nome2 e Nome3 são os nomes dos componentes do grupo.

  No subdiretório `Programas` deverão estar todos os programas fonte e um arquivo `makefile` que compile todo código. Se os programas não compilarem a nota do trabalho será *ZERO*.

  No subdiretório `Texto` deverá estar o arquivo com o seu relatório em formato `ps` ou `pdf`. Se o arquivo não estiver em um destes formatos, a nota do trabalho será *ZERO*.

- Melhorias nos programas serão bem-vindas e podem propiciar bônus adicional no trabalho. Por exemplo, controle da coluna e linha em que ocorre um erro, tratamento de erros para indicar vários erros sintáticos. Uma outra melhoria possível é estender o analisador sintático para aceitar os outros controles de fluxos (`if-else`, `break`, `do-while`, etc) da linguagem C.