Gabriel Tourinho, Matheus Lima, Thiago Santos

Ciência da Computação– Universidade Federal da Bahia (UFBA)

Resumo. *Este artigo descreve o detalha a criação de um compilador para simulação de um robô móvel, na linguagem Robot-L, e vale projeto prático para a disciplina MATA61 - Compiladores.*

Estrutura do compilador

O processo de compilador passa por quatro partes. Recebendo como entrada um código fonte, o analisador léxico gera uma sequência de tokens e passa para o analisador sintático. Durante a análise sintática, também é feita a análise semântica e geração de código.

Em cada produção verificada, é gerado o código intermediário e, nas produções específicas que geram conflitos semânticos, é feita a análise semântica, com o auxílio da tabela de símbolos.

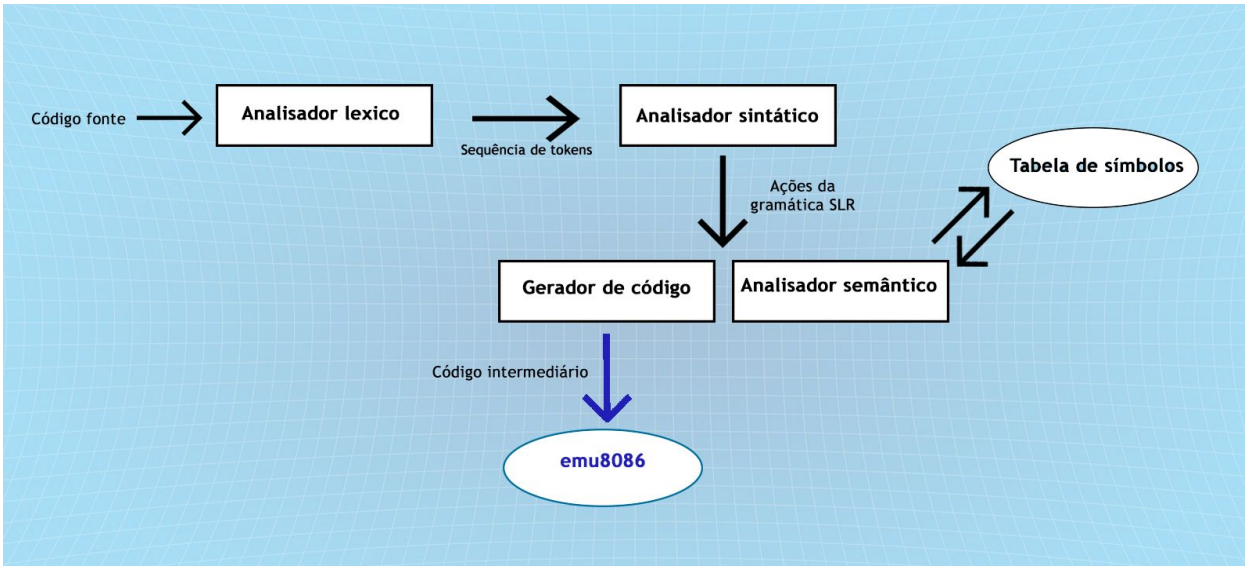

Imagem 1. Fluxo do processo de compilação

Análise léxica

Segundo Alfred V. Aho et al., se tratando da primeira parte de um compilador, a principal função do analisador léxico é ler os caracteres de entrada do programa fonte, agrupá-los em lexemas e produzir como saída uma sequência de tokens para cada lexema do programa fonte.

A partir daí, a primeira decisão de projeto foi como definir um token e quais informações deveriam ser guardadas em cada um. Optamos por criar um objeto chamado Token, que tem tipo, valor, linha e coluna, que seriam as informações necessárias para a implementação do resto do compilador.

Depois disso, a próxima decisão era a de fazer a análise léxica usando expressões regulares ou implementar um autômato. Analisando a linguagem especificada, decidimos que, por não ser muito complexa, seria mais simples implementar a análise léxica baseada num autômato, já que se trata de uma linguagem simples, fazer a análise léxica através de expressões regulares poderia gerar complicações desnecessárias.

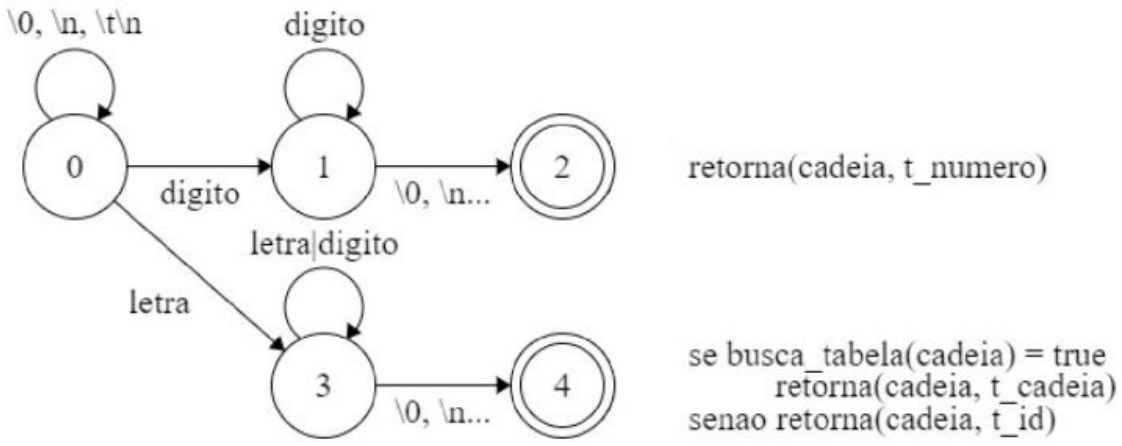

Imagem 2. Autômato usado para a análise léxica

Definido o que seria um token, e como a análise léxica seria feita, o próximo passo era separar as palavras reservadas. A gramática da linguagem tem algumas produções como Condição -> “Robo Pronto” e Condição -> “Lampada Apagada a Direita”, onde cada palavra poderia ser um token separado, ou toda a parte direita poderia ser agrupada em um único token. Algumas linguagens Python e C usam cada palavra como uma palavra reservada, então implementamos dessa forma, e tratamos a formação de tokens compostos na análise sintática.

Além disso, existem basicamente duas formas de tratar as palavras reservadas. Ter um estado final no autômato para cada palavra reservada, e identificá-las separadamente, ou a segunda forma, que é como foi implementado, o analisador léxico busca por um identificador, e, ao terminar, compara o identificador com as palavras na lista de palavras

reservadas. Se o identificador formado estiver nessa lista, gera uma palavra reservada, caso contrário, gera um identificador.

```
if stringId.casefold() in Token.reservadas:
    # verifica se é palavra
    token = Token(Token.reservada, stringId.lower(), self.linhaAtual[self.numLinha], self.numLinha, self.posLinha)
    token.tipo = stringId.lower()
# é identificador
else:
    token = Token(Token.identificador, stringId, self.linhaAtual[self.numLinha], self.numLinha, self.posLinha)
self.tokens.append(token)
```

Imagem 3. Tratamento de palavras reservadas

Outra função do analisador léxico é ignorar espaços em branco, quebras de linha e comentários. Assim, na linguagem Robot-L os comentários começam com '#', e, a partir dele, enquanto estiver na mesma linha, nenhum caractere que vier a seguir gerará erro léxico.

Por fim, a última função do analisador léxico se refere aos erros léxicos. Do ponto de vista do analisador léxico, não há muito o que fazer ao encontrar um erro, além de reportá-lo. Assim, existem dois tipos de erros léxicos: token mal formado e caractere que não faz parte da linguagem. No primeiro caso, o erro indicará a posição do último caractere válido, com linha e coluna, por exemplo, '003a' seguia a regra de formação de um dígito, mas como o 'a' não faz parte de um dígito, o erro vai apontar para a posição do caractere '3'. No segundo tipo de erro, ao encontrar, por exemplo, '!', que é um caractere que não faz parte da linguagem, o erro será na posição do caractere.

Análise sintática e semântica

Se a análise léxica foi feita e não gerou nenhum erro, é gerada uma lista de tokens e passada para a análise sintática. Com isso, o parser verifica se a sequência de tokens passada pelo analisador léxico pode ser gerada pela gramática da linguagem.

As linguagens de programação têm regras bem estruturadas que definem a estrutura sintática dos programas. Para isso, foi gerada uma gramática SLR (1), e o parser é do tipo bottom-up.

```
START -> PROGRAMA

PROGRAMA -> programainicio DEC execucaoinicio CMD fimexecucao fimprograma

DEC -> DECLARACAO DEC
DEC -> ''

DECLARACAO -> definainstrucao identificador como DECAUX

DECAUX -> COMANDO
DECAUX -> BLOCO

BLOCO   -> inicio CMD fim

CMD     ->  BLOCO
CMD     ->  COMANDO CMD
CMD     ->  ''

COMANDO -> ITERACAO
COMANDO -> INSTRUCAO
COMANDO -> LACO
COMANDO -> CONDICIONAL

ITERACAO -> repita numero vezes COMANDO fimrepita

LACO -> enquanto CONDICAO faca COMANDO fimpara

CONDICIONAL -> se CONDICAO entao COMANDO fimse CNDCNL

CNDCNL -> senao COMANDO fimsenao
CNDCNL -> ''
```

```
ESTADO -> pronto
ESTADO -> ocupado
ESTADO -> parado
ESTADO -> movimentando

DIRECAO -> frente
DIRECAO -> SENTIDO

ESTADOLAMPADA -> lampada acesa
ESTADOLAMPADA -> lampada apagada

NUMPASSOS -> numero passos
NUMPASSOS -> ''

SENTIDO -> direita
SENTIDO -> esquerda


CONDICAO -> robo ESTADO
CONDICAO -> DIRECAO robo bloqueada
CONDICAO -> ESTADOLAMPADA a DIRECAO

INSTRUCAO -> mova NUMPASSOS
INSTRUCAO -> vire para SENTIDO
INSTRUCAO -> identificador
INSTRUCAO -> pare
INSTRUCAO -> finalize
INSTRUCAO -> apague lampada
INSTRUCAO -> acenda lampada
INSTRUCAO -> aguarde ate CONDICAO
```

Imagem 4. Gramática SLR para a linguagem Robot-L

A vantagem de usar uma gramática SLR (1) é que, depois de construída a gramática e a tabela de parsing, é fácil fazer as operações necessárias.

Assim, geramos a tabela de parsing no arquivo tabela.py, colocamos as produções da gramática numa variável, e depois disso foi basicamente implementar o algoritmo de shift-reduce.

Para implementar, primeiro atribuímos a sequência de tokens a uma variável entrada e criamos uma classe state, que tem os seguintes atributos:

- tipo
- estado
- code
- valor

O próximo passo é aplicar o shift-reduce seguindo os valores contidos na tabela sintática.

Assim como na análise léxica, ao encontrar um erro, o analisador sintático apenas informa que o erro existe, mas no caso de um erro sintático só é informada a linha do erro.

A análise semântica é a terceira fase do compilador, e consiste em verificar se a entrada está semanticamente correta. Na linguagem desse projeto, são quatro os casos que

podem gerar erros semânticos: duas declarações de instrução com o mesmo nome; chamada para uma instrução não declarada; declarações imediatamente subseqüentes com sentidos diferentes; e após uma instrução ‘mova n’, em que n representa o número de passos, deve ser precedida por uma instrução do tipo ‘aguarde até pronto’.

Para a análise semântica, existem algumas formas de implementação. Optamos por fazer a análise semântica juntamente à análise sintática, com uma função de análise semântica chamada antes de cada reduce do analisador sintático, além de algoritmos auxiliares dentro do shift para guardar valores importantes para a análise.

Além disso, quando é chamada uma função com início ou final “mova para sentido”, comparamos com a instrução anterior e subsequente a chamada para não permitir a chamada de dois sentidos subsequentes, mesmo com profundidade.

A função de análise semântica é sempre chamada antes de cada reduce do sintático, pois, por exemplo, quando vamos fazer uma redução do tipo “instrução -> vire para sentido”, sabemos que a próxima instrução está guardada na entrada. Então, verificamos o sentido da instrução e comparamos com o início da entrada para verificar se há outra chamada de movimento com sentido oposto.

Os erros semânticos encontrados são indicados com a linha no código onde o erro se encontra, além do tipo do erro. Além dos quatro possíveis casos onde podem existir erro que estão na especificação do projeto, implementamos um caso onde gera um warning. No caso de uma instrução que começa, por exemplo, com ‘vire para esquerda’, e termine com ‘vire para direita’, o loop dessa instrução pode gerar um erro de ‘vire para esquerda’ seguido de ‘vire para direita’. Então, se uma instrução é definida de modo que a primeira instrução é ‘vire para x’ e termina com ‘vire para y’, se x e y forem sentidos opostos, gerará um warning.

Geração de código

Na geração de código, o compilador deve traduzir o código fonte em um código intermediário independente, que depois será usado para gerar o código que pode ser entendido pelo computador.

Se o código fonte passar pelas três primeiras fases do compilador sem erros, começa a geração de código. Para isso, de início traduzimos as produções da gramática para código intermediário, e com isso, a cada reduce feito na análise sintática, é chamada a função que faz a geração de código.

```
def gerarCodigo(pilha,regra,contIf,contElse,contWhile,contbusy,contiter,contaguarde):
    code=''
    delay='mov cx,50\nbusy:\nloop busy\n'
    moverFrente='mov al,1\nout 9,al\nmov cx,50\nbusy:\nloop busy\n'
    moverDireita='mov al,3\nout 9,al\nmov cx,50\nbusy:\nloop busy\n'
    moverEsquerda='mov al,2\nout 9,al\nmov cx,50\nbusy:\nloop busy\n'
    pare='mov al,0\nout 9,al\nmov cx,50\nbusy:\nloop busy\n'
    examine='mov al,4\nout 9,al\nbusy: in al, 11\ntest al, 00000001b\njz busy\nin al,10\n'
    bloqueio='cmp al,255\nje label\n'
    lampadaAcesa='cmp al,7\nje label\n'
    lampadaApaguada='cmp al,8\nje label\n'
    acendalampada='mov al,5\nout 9,al\nmov cx,50\nbusy:\nloop busy\n'
    apaguelampada='mov al,6\nout 9,al\nmov cx,50\nbusy:\nloop busy\n'
```

Imagem 5. Esquema de tradução usado na geração de código

Para gerar o código recebemos a produção, que está dentro da variável ‘regra’, e alguns contadores como ‘contif’ e ‘contelse’. Além desses contadores, a geração de código também recebe a pilha para que possamos pegar o ‘valor’ e o ‘code’, que são atributos da classe state.

O atributo ‘code’ é usado para guardar o código intermediário da produção recebida e o valor é usado para casos como o ‘NUMPASSOS->numero passos’, que é necessário guardar o valor do token número no state ‘NUMPASSOS’.

Usamos a mesma estratégia que as gramáticas S-atribuídas, que o pai herda os valores dos filhos. Isso se torna evidente em casos como na produção ‘DECLARACAO->definainstrucao identificador como DECAUX’, pois o ‘code’ do state ‘DEACAUX’ é herdado por ’DECLARACAO’.

Existem também casos em que temos que modificar o valor de code, que será herdado, como no caso de ‘LACO->enquanto CONDICAO faca DECAUX fimpara’.. Neste caso, iremos concatenar a string ‘label: ‘ no inicio do valor de ‘code‘ de ‘DECAUX’, que é concatenado com ‘code’ de ‘CONDICAO’.

No final da análise sintática teremos todo o código intermediário armazenado no state PROGRAMA.

```
elif 'acc' == actions:
    print('Sem erros sintáticos')
    if countsemantico == 0:
        print('Sem erros semânticos\n')
        open('programa.asm','w').write(code)
    entrada=entrada[1:]
else :
    print('Erro sintático na linha: ',entrada[0].numLinha+1)
    break
```

Imagem 5. Geração de código sendo feita junto com a análise sintática e semântica

Exemplo de um programa

Para este exemplo usaremos o código contido na especificação do projeto.

```
PROGRAMAINICIO
    definainstrucao trilha como
    inicio
        mova 3 passos
        aguarde ate robo pronto
        vire para esquerda
        apague lampada
        vire para direita
        mova 1 passos
        aguarde ate robo pronto
    fim
    EXECUCAOinicio
        repita 3 vezes trilha fimrepita
        vire para direita
    fimEXECUCAO
fimprograma
```

Imagem 6. Programa de exemplo

Depois da análise léxica, são gerados os seguintes tokens

```
'programainicio', 'definainstrucao', 'identificador', 'como', 'inicio', 'mova', 'numero', 'passos',
'aguarde', 'ate', 'robo', 'pronto', 'vire', 'para', 'esquerda', 'apague', 'lampada', 'vire', 'para',
'direita', 'mova', 'numero', 'passos', 'aguarde', 'ate', 'robo', 'pronto', 'fim', 'execucaoinicio',
'repita', 'numero', 'vezes', 'identificador', 'fimrepita', 'vire', 'para', 'direita', 'fimexecucao',
'fimprograma'
```

Imagem 7. Tokens geradas a partir do programa exemplo

Após passar os tokens pela análise sintática e semântica, é gerado o seguinte código intermediário:

```
#start=robot.exe#
mov cx,50
busy:
loop busy
mov cx, 3
iteracao0: push cx
call trilha
pop cx
loop iteracao0
mov al,3
out 9,al
mov cx,50
busy16:
loop busy16
mov al,0
out 9,al
mov cx,50
busy17:
loop busy17
hlt
trilha proc
mov al,1
out 9,al
mov cx,50
busy0:
loop busy0
mov al,0
out 9,al
mov cx,50
busy1:
loop busy1
mov al,1
```

```
out 9,al
mov cx,50
busy2:
loop busy2
mov al,0
out 9,al
mov cx,50
busy3:
loop busy3
mov al,1
out 9,al
mov cx,50
busy4:
loop busy4
mov al,0
out 9,al
mov cx,50
busy5:
loop busy5
mov cx,50
busy6:
loop busy6
aguarde0: in al,11
test al,00000010b
jnz aguarde0
mov al,2
out 9,al
mov cx,50
busy7:
loop busy7
mov al,0
out 9,al
```

```
mov cx,50
busy8:
loop busy8
mov al,6
out 9,al
mov cx,50
busy9:
loop busy9
mov al,0
out 9,al
mov cx,50
busy10:
loop busy10
mov al,3
out 9,al
mov cx,50
busy11:
loop busy11
mov al,0
out 9,al
mov cx,50
busy12:
loop busy12
mov al,1
out 9,al
mov cx,50
busy13:
loop busy13
mov al,0
out 9,al
mov cx,50
busy14:
```

```
loop busy14
mov cx,50
busy15:
loop busy15
aguarde1: in al,11
test al,00000010b
jnz aguarde1
ret
trilha endp
```

Imagens 8, 9 e 10. Código intermediário gerado

Referências

Alfred V. Aho et al. - Compilers - Principles, Techniques, and Tools. 2nd Edition. Pearson - Addison Wesley, 2006

Keith Cooper, Linda Torczon - Engineering: A compiler. Morgan Kaufmann; 2 edition, 2011

Charles Fischer et al.- Crafting A Compiler. Pearson, 2009