Descrição do Trabalho Prático 03 de Compiladores

Eduardo G. R. Miranda

Julho 2024

1 Introdução

O trabalho prático (TP) deve ser realizado em dupla e consiste na implementação de geração de código intermediário e análise semântica fazendo uso do Analisador Sintático feito no trabalho prático 02.

2 Proposta de TP

Conforme mencionado, o TP proposto consiste em implementar um analisador sintático que gere o código intermediário para ser executado na máquina virtual.

2.1 Sobre a Geração de Código intermediário

O programa resultante deve possibilitar que o usuário forneça um arquivo .pas, e que seja feita inicialmente a análise léxica e juntamente com a segunda etapa de análise sintática o código intermediário seja gerado e retornado de acordo com como o programa precisa funcionar.

2.1.1 Tuplas e Lista de tuplas

Assim como já foi feito na parte 04 (Interpretador) o retorno desta parte é uma lista de tuplas conforme pode ser visto na especificação.

Por exemplo na parte relacionada a entrada :

```
# comandos de IO
<ioStmt> -> 'read' '(' 'IDENT' ')' ';'
```

Neste caso, se houver os tokens de leitura virem na ordem correta, o lexema do 'IDENTIFICADOR'(Variável) deve ser salvo e deve ser gerada uma tupla de entrada no seguinte formato: ("CALL","SCAN","LEXEMA","TIPO").

Já na parte relacionada a saída de informações:

```
# comandos de IO
<ioStmt> -> 'write' '(' <outList> ')' ';' ;
```

Deve ser gerado tuplas no formato: ("CALL","READ","VALOR",None). Todavia a função 'outlist' possui a chamada da função 'restoOutList' irá gerar recursão. Para tratar esta recursão e retornar de forma correta os comandos deve-se retornar uma tupla na função 'out', retornar uma lista de tuplas nas funções 'outList' e 'restoOutList' e concatenar as listas retornadas de forma recursiva, ou adicionar uma única tupla a lista atual.

2.1.2 Gerador de Labels e Variáveis temporárias

Para realizar o controle de fluxo do ponteiro de execução no código intermediário se faz necessário criar tuplas do tipo label: ("label", "NomeLabel", None, None). Nos comandos que exigem controle de fluxo (if, for, while, ...) será necessário adicionar tuplas de controle para mudar o ponteiro de execução para o lugar certo. Portanto um gerador de label conforme a necessidade precisará de ser criado.

Da mesma maneira que as labels serão criadas, para realizar operações também pode ser necessário o uso de variáveis auxiliares, então também se fará necessário o uso de gerador de nomes de variáveis temporárias.

2.1.3 Comandos de controle de fluxo

Para lidar com os comandos de fluxo (if, while e for) será necessário dividir o problema para garantir que o código intermediário seja montado no formato correto.

```
<forStmt> -> 'for' <atrib> 'to' <endFor> 'do' <stmt> ;
```

No exemplo acima temos a regra da gramática do comando 'for', temos 3 partes principais 'atrib', 'endFor' e 'stmt'. Essas partes representam respectivamente, a lista do código intermediário da atribuição da variável de controle, a lista do código intermediário usada para verificar se deve-se sair do for ou não, e a lista do código intermediário que deve ser executada repetidas vezes de acordo com o comando for.

Dado que possuímos essas 3 listas de código intermediário o que é necessário agora é, posicionar labels que foram geradas pelo gerador de label e fazer uso dos comandos IF e JUMP para fazer com que o código que deve ser executado mais de uma vez, seja executado repetidamente até que a condição de saída seja verdadeira.

Para os outros comandos (while e if) o princípio a ser seguido é o mesmo.

2.1.4 Regra Declarations

A regra 'declarations' também é recursiva assim como a regra 'outlist' apresentada na Subseção 2.1.1, esta regra deve ser implementada de forma específica, visto que só se terá conhecimento do tipo da variável após ler todas as variáveis, devem ser guardadas todos os lexemas destas variáveis que serão retornados de forma recursiva, e após isso montar tuplas de atribuição de acordo com o tipo da variável.

O comando: 'a1,a2,a3:integer;'. Será transformado em:

```
[
    ("=","a1",0,None),
    ("=","a2",0,None),
    ("=","a3",0,None)
]
```

Já o comando: 'a1,a2,a3:string;'. Será transformado em:

```
[
    ("=","a1",'',None),
    ("=","a2",'',None),
    ("=","a3",'',None)
]
```

O mesmo deve ser feito para o tipo float, porem o valor de inicialização será o número 0.0.

2.1.5 Regra expr e restante de sua árvore

Esta parte da gramática possui suas peculiaridades:

Por exemplo, como é feita uma árvore recursiva e que determina a precedência das operações a serem feitas os retornos que serão feitos depende de resultados anteriores, além de que para fazer cálculos também se faz necessário saber os resultados anteriores.

Então o exemplo mais básico de funcionamento conforme código disponibilizado seria o da função fator():

```
atual = self.l.lexema
if self.l.token_atual == enumTkn.tkn_numFloat:
    self.consome(enumTkn.tkn_numFloat)
    return (False, [], atual)
elif self.l.token_atual == enumTkn.tkn_var:
    verifica = self.controle.verifica_simbolo(atual)
    if not verifica:
        msg = "Variavel %s não foi declarada." % atual
        raise ErroSintatico((self.l.linha, self.l.coluna), msg)
    self.consome(enumTkn.tkn_var)
    return (True, [], atual+verifica)
elif self.l.token_atual == enumTkn.tkn_abrePar:
        self.consome(enumTkn.tkn_abrePar)
        left, lista, res = self.atrib()
        self.consome(enumTkn.tkn_fechaPar)
        return (False, lista, res)
```

Neste exemplo a primeira parte verifica qual o tipo de token, caso seja um token numérico (float) se retorna uma lista de comandos vazios e qual o valor

numérico que deve ser utilizado na expressão. A segunda parte verifica caso seja um variável, se for verdade se faz a checagem se esse simbolo está presente entre as variáveis que foram declaradas previamente, caso isto seja verdade se retorna uma lista de comandos vazios e o nome da variável que deve ser utilizada na expressão. Por fim a parte de expressão chama recursivamente o início da árvore para resolver uma determinada quantidade de comandos com maior prioridade começa com um parenteses; nesta parte pode ser visto que é retornada uma lista de comandos a serem executados além de uma variável (provavelmente temporária) que deve ser retornada em conjunto para resolução da expressão.

Um segundo exemplo seria a funcaoOR:

```
leftValue, listaComandos, resultado = self.functionAnd()
leftValueb, listaComandosb, resultadob = self.restoOr(resultado)
listaComandos.extend(listaComandosb)
return leftValue and leftValueb, listaComandos, resultadob
```

Neste caso recebemos uma lista de comandos da funcaoAnd além da variável que deve ser utilizada para serem feitos os cálculos, essa lista de comandos e essa variável vão vir por alguma parte inferior da árvore de recursão. E como as operações precisam de 2 operadores para serem resolvidas essa variável que guarda o resultado será passada como parâmetro para a função de restoOr para que o código intermediário da função or seja feito utilizando as duas variáveis corretas. Por fim se estende a lista de comandos retornados da chamada da primeira função com a da segunda e é retornado a nova variável.

Por fim a função restoOr tem o seguinte funcionamento:

```
def restoOr(self, resultant):
    if self.l.token_atual == enumTkn.tkn_or:
        opr = self.l.lexema
        self.consome(enumTkn.tkn_or)
        leftValue, listaComandos, resultado = self.functionAnd()

        novotemp = self.controle.geraTemp() # ficara o resultado do or

        listaComandos.append((opr, novotemp, resultado, resultant))
        leftValueb, listaComandosb, resultadob = self.restoOr(novotemp)
        listaComandos.extend(listaComandosb)

        return False, listaComandos, resultadob
    else:
        return True, [], resultant
```

Começando pelo funcionamento mais simples, caso a operação que está sendo realizada não seja um or (caso do else), é somente retornado o valor para a

parte superior da árvore de recursão e uma lista de comandos vazia, que se estendida não possui efeito algum. Porém caso esteja-se realizando uma operação or de fato é consumido o token_or e para a segunda variável que deve ser comparada é feita uma chamada a functionAnd, para que esta função possa "descer a árvore de recursão" até chegar a função fator e retornar com qual variável deve ser resolvido o a expressão lógica (resultant or resultado)[ressalta-se que a variável resultant já veio de uma "descida de árvore de recursão e foi passada por parâmetro para a função, assim como a variável resultado também o fará, estas podem ser variáveis ou valores numéricos simples, tanto quanto podem ser resultados de outras operações feitas primeiro devido a precedência descrita pela gramática que geraram "variáveis temporárias" para guardarem esses valores].

Após termos as duas variáveis que serão utilizadas para realizar a operação OR criamos o código intermediário da operação que deve ser realizada e chamamos recursivamente o restoOR para caso haja mais operações do tipo OR a serem feitas estendemos essa lista e a retornamos juntamente com a nova variável que deve ser considerada pela função "pai".

As demais funções relacionadas a expressão funcionam de maneira similar a alguma destas que foram mencionadas.

2.1.6 Demais regras da gramática

Como já foi mencionado na segunda parte, a gramática gera uma árvore recursiva.

Portanto para gerar a lista de comandos do código intermediário basta controlar o retorno das tuplas e fazer a concatenação correta das listas.

Em termos gerais, o trabalho consiste em gerar uma lista 'local' de uma regra com suas determinadas tuplas, retornar esta e concatenar com o que já se tem. Ao fazer isto a árvore gramatical construída pela chamada das funções fará com que a concatenação destas listas fique na ordem em que o código intermediário deve ser executado.

3 Algumas Dicas

Durante o desenvolvimento, é importante não se perder nos detalhes. Portanto, é recomendado que os alunos comecem o desenvolvimento implementando as funcionalidades básicas. Só depois de garantir que as funcionalidades básicas estão funcionando conforme planejado, os alunos devem considerar a implementação de melhoramentos e funcionalidades adicionais. Também recomenda-se que trechos mais complicados do código sejam acompanhados de comentários que esclareçam o seu funcionamento/objetivo/parâmetros de entrada e resultados.

O TP também será avaliado em termos de:

- Modularização;

- Legibilidade (nomes de variáveis significativos, código bem formatado, uso de comentários);
- Consistência (formatação uniforme);
- Otimização do Código (Uso das estruturas de dados mais eficientes, eliminação de redundâncias, etc...);
- Arguição oral sobre a solução desenvolvida com o grupo com o grupo (**Fator Multiplicativo** ao demonstrar o domínio do código e do problema tratado).

3.1 Funcionalidade

Ao fornecer um programa da linguagem MiniPascal espera-se que o código intermediário tenha sido gerado corretamente de forma a executar corretamente na máquina virtual criada para a parte 04 do interpretador o que estaria codificado na linguagem original.

3.2 Submissão

Além de serem submetidos via link no Portal Didático da disciplina, os TPs também devem ser enviados para o endereço eletrônico `eduardomiranda@cefetmg.br`, tendo como assunto TP_LFA_2024_1. No corpo do email deve aparecer o nome completo de cada integrante do grupo e um arquivo .zip com o programa e o manual de utilização.