**PCS2056 – Linguagens e Compiladores**

**4ª parte do projeto**

**Ambiente de execução**

**Alan Raso – 804151**1. Introdução

Ambiente de execução é uma estrutura que oferece todo o suporte necessário à execução do programa escrito na linguagem elaborada. Ele é composto de uma interface com o sistema operacional, gerenciamento de memória, com tratamento de sub-rotinas e parâmetros, além de rotinas para o tratamento de recursos que a linguagem oferece, como expressões aritiméticas.

O gerenciamento de memória depende da linguagem, mas, em comum a todas, deve-se reservar o espaço de memória necessário para a execução do programa. Algumas linguagens permitem que isso seja feito em tempo de compilação, ou seja, todos os elementos e procedimentos da linguagem em um tamanho fixo possível de ser calculado. Isso é chamado de alocação estática. Outras linguagens disponibilizam ferramentas que inviabilizam este tipo de alocação, como por exemplo, recursão.

Para o caso citado, é utilizada uma pilha e é reservada uma área de memória a cada bloco. Esta pilha é preenchida com os chamados registros de ativação, que são estruturas de dados que possuem áreas para as seguintes informações: objetos declarados em uma função, os parâmetros desta função, endereço de retorno, ponteiros para outros registros de ativação e valor de retorno. A este caso denomina-se *alocação automática.*

Existe, ainda, um caso em que não é possível aplicar nenhum dos tipos de alocação citados, em que é utilizada alocação dinâmica. Para eles, utiliza-se uma área de memória chamada de *“heap”*, que é utilizada conformea demanda.

Existem várias formas de projetar um ambiente desta forma, principalmente ao se variar as estruturas de dados utilizadas para o tratamento de cada elemento da linguagem. Para o caso específico deste projeto, o ambiente de execução será composto de tabelas para o tratamento de escopos, sendo uma tabela para cada um. Os escopos de hierarquia mais baixa serão apontados pelo escopo acima dele, e informações como variáveis, seus tipos e memória ocupada serão armazanados nela. Para o tratamento de expressões, duas pilhas serão utilizadas, uma de operandos e a outra de operadores, e rotinas para tratar a emissão de valores, inclusive temporários que possam ser necessários.

# 2. Instruções da linguagem de saída

A linguagem de saída é a mesma que foi trabalhada na disciplina de Fundamentos da Engenharia da Computação (PCS2024), a MVN.

Esta linguagem tem 8 registradores que são preenchidos e consultados pelas instruções existentes na linguagem, fazendo o programa funcionar como esperado. De um modo geral, as instruções tem um formato simples: um comando, seguido por espaço e um operando, ambos números hexadecimais. Para melhorar a legibilidade, foram criados mnemônicos para os comandos. Os comandos listados abaixos são seguidos de um exemplo e de uma breve descrição.

## 2.1. Desvio incondicional: “JP 0040”

O desvio incondicional carrega o valor do operando para o registrador chamado IC, que indica o valor da próxima instrução que será lida na memória. Assim, é possível pular para uma parte específica do código. É possível também a utilização de rótulos para melhorar a legibilidade.

## 2.2. Desvio se o acumulador é zero: “JZ 0040”

Esta instrução é similar ao desvio incondicional, com a única diferença de que o valor do operando é carregado no registrador IC apenas se o registrador acumulador (AC) tem o valor zero nele.

## 2.3. Desvio se o acumulador é negativo: “JN 0040”

Instrução similar a 2.2, mas apenas é executada se o valor no acumulador é negativo.

## 2.4. Depostia uma constante no acumulador: “LV 1F00”

Carrega o valor do operando passado na instrução para o acumulador. Muitas instruções utilizam o valor armazenado no acumulador para serem executadas. Os *jumps* apresentados em 2.2 e 2.3 são exemplos disso.

## 2.5. Soma: “+ 1002”:

Realiza a soma entre o valor atual no acumulador e o operando, e guarda o resultado no acumulador.

## 2.5. Soma: “+ 1002”:

Realiza a soma entre o valor atual no acumulador e o valor guardado no endereço de memória passado pelo operando da instrução, e guarda o resultado no acumulador.

## 2.6. Subtração: “- 0004”:

Realiza a subtração entre o valor atual no acumulador e o valor guardado no endereço de memória passado pelo operando da instrução, e guarda o resultado no acumulador.

## 2.7. Multiplicação: “\* 0002”:

Realiza a multiplicação entre atual no acumulador e o valor guardado no endereço de memória passado pelo operando da instrução, e guarda o resultado no acumulador.

## 2.8. Divisão: “/ 0002”:

Realiza a divisão entre atual no acumulador e o valor guardado no endereço de memória passado pelo operando da instrução, e guarda o resultado no acumulador.

## 2.9. Memória para acumulador: “LD 0006”:

Carrega para o acumulador o valor presente no endereço de memória passado pelo operando da instrução.

## 2.10. Acumulador para a memória: “MM 1002”:

Carrega para o endereço de memória especificado no operando o valor atual do acumulador.

## 2.11. Desvio para subprograma: “SC 2000”:

Carrega o valor atual o registrador IC para o registrador RA, responsável por armazenar o endereço de retorna de uma sub-rotina. Além disso, carrega no registrador IC o valor do operando. Isso significa que a próxima instrução será a do endereço passado, e ao final da sub-rotina, o programa retornará onde estava (valor carregadono RA). Assim como nos jumps, é possível rotular.

## 2.12. Retorno de subprograma: “RS 0002”:

Carrega o valor passado pelo operando no acumulador e carrega o valor do registrador RA no registrador IC, ou seja, a próxima instrução será aquela que foi gravada na chamada da sub-rotina em questão.

## 2.13. Parada: “Rot1 HM Rot1”:

Todas as instruções, com exceção dos *jumps* e chamadas e retorno de sub-rotina incrementam o registrador IC em 1, ou seja, passam para a próxima instrução. A instrução de parada é utilizada para que o programa pare, guardando no valor do registrador IC o valor da própria instrução, que geralmente é rotulado para ele mesmo.

## 2.14. Entrada: “GD Disp”:

Aguarda a entrada deum valor por um dispositivo de entrada e carrega esse dado no acumulador.

## 2.15. Saída: “GD Disp”:

Apresenta o dado do acumulador como saída.

3. Pseudoinstruções da linguagem de saída

* **@:** a instrução seguinte é definida pelo operando dessa pseudoisntrução;
* **K:** define um valor constante que é o operando desta pseudoinstrução;
* **$:** realiza uma reserva de dados com o tamanho definido pelo operando;
* **#:** define o final físico do texto fonte por meio de rótulo.

4. Características gerais

## 4.1. Gerenciamento de memória

O gerenciamento de memória será de alocação automática, e portanto realizado por meio de registros de ativação.

## 4.2. Escopos

Os escopos da linguagem, apresentados por meio de chaves, serão representados em tabelas, uma para cada escopo. O código abaixo numera alguns escopos:

func main ()() { **(0,0)**

int a;

a := 2;

se (a = 2) { **(1,1)**

printf(“O valor é 2!”);

} senao { **(1,2)**

se (a < 2) { **(2,3)**

printf(“O valor é menor do que 2!”);

} senao { **(2,4)**

pritnf(“O valor é maior do que 2!);

}

}

}

Os números n,m representam o nível hierárquico do escopo e a numeração dele, respectivamente. O nível hierárquico corresponde basicamente ao número de escopos “pai” que ele possui, enquanto a numeração é, começando do 0, sequencial conforme o código avança. O número de tabelas geradas é o númeo do último escopo numerado mais 1 (m+1).

O conteúdo dessa tabela é composto por informações relevantes ao escopo, como as variáveis declaradas, seus tipos e o endereço relativo ao endereço de início do escopo (offset). Além disso, existem ponteiros para as tabelas dos escopos “filhos”, bem como um campo que armazena o valor calculado dp tamanho na memória que cada um ocupa. Outros dados relevantes podem ainda ser inseridos, como, por exemplo, o tamanho que a variável ocupa, informação útil para arrays.

A ideia é que quando uma variável for encontrada, ela seja buscada na tabela relativa ao escopo dela. Caso não seja encontrada, será buscada na tabela do ancestral, e assim por diante.