Universidade Federal do Piauí – Departamento de Ciência da Computação

Compiladores – 2024.2 - Prof. Glauber Dias Gonçalves

Trabalho final da terceira unidade – 2024.2 Valor 10,00 pontos

1. Desenvolvimento

O objetivo é desenvolver um compilador para a linguagem LPMS (linguagem de programação muito simplificada) descrita nessa especificação. O trabalho pode ser desenvolvido individualmente ou em equipes de no máximo três membros.

O trabalho está organizado em três etapas. Cada etapa consiste na implementação incremental do compilador até alcançar a 3ª etapa onde o compilador deve estar completo e funcional. O compilador de cada etapa deve ler da entrada padrão um arquivo com o código fonte do programa a ser compilado. Contudo, os resultados do compilador são diferentes em cada etapa:

- 1ª etapa: análise léxica (*scanner*). Imprimir na saída padrão cada token (i.e., tupla <token_type, token_atributes>) encontrado no arquivo de entrada, semelhante à Tarefa 3.
- 2ª etapa: análise sintática com análise semântica (*parser*). Imprimir na saída padrão uma representação da árvore sintática abstrata (AST) aninhada com parênteses ou erro sintático/semântico indicando a linha onde ocorreu.
- 3ª etapa: geração de código. Gerar um arquivo com código intermediário e, opcionalmente, código destino executável.

Em cada etapa deve ser disponibilizado o código fonte de compilador e um arquivo readme.txt com informações sobre como executá-lo e uma breve descrição das referências utilizadas em seu desenvolvimento.

Para o desenvolvimento do projeto, ferramentas para geração automática de scanners e parsers como PLY, Lex/Yacc, Flex/Bison, LLVM ou ANTLR podem ser utilizadas. Do mesmo a linguagem adotada para a implementação do compilador fica a critério da equipe, podendo ser Java, Python ou C/C++.

A linguagem fonte do projeto apresenta diversas limitações e possui somente fins acadêmicos. LPMS será especificada informalmente neste documento. O primeiro desafio do projeto é transformar esta descrição em uma especificação formal, através de uma gramática livre de contexto (GLC).

O compilador deverá traduzir programas escritos em Java Simplificado para uma linguagem intermediária, podendo ser código de três endereço (*three address code*) ou código intermediário específico como JASMIM para Java Virtual Machine ou LLVM. A geração de código de máquina, ou seja, executável a partir da linguagem intermediária é opcional. Contudo, receberá nota máxima apenas trabalhos que gerarem código de máquina.

2. Entregas e avaliação

Haverá uma data de entrega para cada etapa do trabalho. Cada entrega consiste na apresentação da etapa desenvolvida ao professor/monitor por equipe/discente e submissão dos códigos fontes no Sigaa. Cada etapa vale 3,0 pontos, sendo 1,0 ponto

referente a uma pergunta dirigida a cada membro da equipe na apresentação. A 3ª etapa tem um ponto adicional para a geração de código executável.

Assim, o calendário da disciplina em janeiro tem a seguinte organização, considerando entregas e avaliações:

- 07/ janeiro (3ª feira): atendimento a dúvidas específicas por grupo de trabalho.
- 09/ janeiro (5ª feira): apresentação da 1ª etapa individualmente para professor/monitor, valendo 3,0 pontos e submissão no Sigaa até 23:59.
- 14/janeiro (3ª feira): atendimento a dúvidas específicas por grupo de trabalho.
- 16/janeiro (5ª feira): apresentação da 2ª etapa individualmente para professor/monitor, valendo 3,0 pontos e submissão no Sigaa até 23:59.
- 21/janeiro (3ª feira): apresentação da 3ª etapa individualmente para professor/monitor, valendo 4,0 pontos (1,0 ponto especificamente para a geração de código executável) e submissão no Sigaa até 23:59.

3. A linguagem LPMS

LPS é sensível (*case-sensitive*) em relação a letras maiúsculas e minúsculas nos nomes das variáveis, funções e palavras reservadas. (i.e. “variable”, “VARIABLE” e “VaRiAbLe” representam identificadores diferentes).

A linguagem não suporta declarações de funções, mas possui funções nativas (*built-in*), utilizadas para realizar entrada e saída de dados (*print* e *scanf*) que serão a única forma de se comunicar com o mundo exterior. Adicionalmente, nenhuma característica de outra linguagem deve ser reconhecida pelo compilador de LPMS.

3.1 Tipos e operadores

LPMS é fortemente tipado e dá suporte aos seguintes tipos: inteiro, real, lógico e caracteres. Os únicos operadores disponíveis na linguagem são:

• ‘!’: negação, inverte o valor booleano ao qual foi aplicada – unário;
• ‘-’: menos unário, inverte o valor inteiro ou real ao qual foi aplicada – unário;
• ‘+’: soma – binário;
• ‘-‘: subtração – binário;
• ‘*’: multiplicação – binário;
• ‘/’: divisão inteira e real – binário;
• ‘==’: comparação, checa se os operadores são iguais – binário;
• ‘!=’: comparação, checa se os operadores são diferentes – binário;
• ‘>=’: maior ou igual que – binário;
• ‘<=’: menor ou igual que – binário;
• ‘>’: maior que – binário;
• ‘<’: menor que – binário;
A hierarquia de precedência de operadores da linguagem C/C++ deve ser utilizada.

3.2 Identificadores

São os nomes utilizados para identificar variáveis e constantes no programa. Identificadores podem ter qualquer tamanho maior que zero e devem obrigatoriamente

começar com uma letra. Após isso, podem conter qualquer letra ou digito ou o símbolo underscore (_).

Dentro do mesmo escopo, variáveis não podem ter o mesmo nome de outras variáveis ou funções. Identificadores nunca podem ter o mesmo nome de uma palavra reservada da linguagem (if, while, etc.).

3.3 Variáveis e Constantes

Variáveis são posições de memória que guardarão dados do programa. Em LPMS, toda variável tem o tipo definido na sua declaração, que não poderá ser alterado. Constantes e variáveis podem ser definidas de maneira intercalada.

Para declarar uma variável, atribui-se um tipo a um identificador válido. O valor padrão da variável definida desta forma será 0 (zero) para inteiros e reais, False para as variáveis do tipo booleano e vazia (“ ”) para caracteres. A declaração de variável pode ocorrer em qualquer local do código, desde que a variável não seja utilizada antes de sua declaração.

Exemplo:

```
const PI = 3.14;
int INT2, FLOAT, inteiro1;
str string1, string2;
const flag = true;
float x;
const IES = “UFPI”;
```

Como se pode observar, declarações sequenciais são permitidas, em uma mesma linha, porém apenas quando as variáveis forem de um mesmo tipo.

3.4 Funções nativas

O compilador deverá reconhecer previamente um conjunto de procedimentos nativos para implementação de entrada e saída do usuário.

- *print*: envia saída para o console. A cada chamada também é enviada uma quebra de linha, exceto entre os itens de print (lista de expressões). A lista de expressões deve ser separada por vírgulas.

Sintaxe de chamada: print (<lista_expressao>) ;

Exemplo:

```
print (“Hello ”, “World!”);   // Hello World!
print (“1 + 1 = ”, 1+1);      // 1 + 1 = 2
print (soma);                 // imprime o valor da variável soma
```

- *input*: requisita a entrada de números inteiros, reais e strings a partir do teclado. O valor fornecido é armazenado na variável usada na chamada. O comando de leitura interrompe a execução do programa até que o usuário digite um valor (o valor é confirmado após pressionar a tecla “<Enter>”).

Sintaxe de chamada: input(<lista de variáveis>);

Exemplo:

print (“Digite um número”);
print (“Digite dois números”);
print (“Soma: ”, a+b+c);

3.5 Comandos de controle

O compilador deve reconhecer os seguintes comandos de controle de execução: if-else, while. Esses comandos sempre devem ser delimitados por { e } para indicar início e fim de blocos.

IF-ELSE sintaxe:

if (<teste_logico>){
...
} [else{
...
}]; //Note que a parte do else é opcional

WHILE sintaxe:

while (<condição>){
...
[break]
...
}

3.6 Bloco principal

Um programa é composto por um bloco principal iniciado pela palavra-chave *Program* seguido de um identificador representando o nome do programa (mesmas regras para identificador) e delimitações { ... } para indicar início e fim do bloco principal. Todo o programa deve ser escrito em um arquivo de texto sem formatação com a extensão com o nome do identificador e extensão “.lps”. A identação de sentenças dentro do código não é obrigatória.

Exemplos:

```
Program Fatorial {
    int fat, res;
    res = 1;
    input(fat);
    while( fat > 1 ){
        print (res);
        res = res * (fat - 1);
    }
    print("Resultado final:", res);
}
```

```
Program Maior_Menor {
    float a, b;
    input(a, b);
    if (a > b){
        print("Maior: ", a);
    }else{
        If( a < b){
            print("Maior: ", b);
        }else{
            print("Iguais!");
        }
    }
```