Universidade Federal Rural de Pernambuco
Disciplina: Compiladores
Professor: Pablo Azevedo Sampaio
Semestre: 2016.2
Última alteração: 23/11/2016

Projeto de Compiladores – Segunda Parte

O objetivo desta segunda parte do projeto é estender a primeira parte do projeto acrescentando as etapas seguintes ao compilador **Doxa**:

- *geração da árvore sintática*,
- *análise semântica,*
- e *geração de código* (*intermediário* ou *final*).

As três seções deste documento detalham os requisitos para cada etapa.

Porém, há um requisito básico (e simples de implementar) para seu compilador que é o de **rodar em linha de comando**. Obrigatoriamente, o projeto deverá permitir que o compilador completo seja executado com um comando nesta forma:

```
> java <Nome da classe principal> arquivoEntrada.dox
```

No exemplo acima, "*arquivoEntrada.yld*" representa algum arquivo fonte de **Doxa** localizado no diretório atual. Como resultado, o programa deve gerar um arquivo de nome similar (e.g., *"arquivoEntrada.asm"*) com o código intermediário ou final resultante.

1. Geração da Árvore Sintática

A análise sintática "identifica" a *árvore sintática* correspondente ao código fonte. Porém, os algoritmos vistos não constroem automaticamente uma estrutura de dados com essa árvore. O que vocês devem fazer é justamente construir uma estrutura de dados que represente esta árvore em memória.

Foi disponibilizado no site[1] um *projeto base* que contém um parser de **Doxa** no CUP bem como as classes para cada construção da linguagem (e.g. classes para "if-else", expressões, etc). Vocês devem estender este projeto fazendo o parser instanciar as classes dadas da forma adequada. Neste caso, saibam que:

- tanto o parser como o as classes da árvore podem requerer algumas correções e ajustes;
- em especial, falta incluir no CUP as ações de instanciação destas classes (que é a criação da árvore propriamente dita);
- você devem adaptar o lexer de vocês (feito no JFLex) para usar com este parser (vejam roteiro no site).

Apesar de não ser estritamente necessária para fazer a compilação, esta etapa de criação da árvore será obrigatória na disciplina, valendo **15%** da 2ª VA.

[1] http://sites.google.com/site/profpablosampaio/disciplinas/compiladores/projeto

2. Análise Semântica

Como visto nas aulas, o **analisador semântico** faz verificações e reúne informações sobre o código para serem usadas nas etapas de geração de código. Dividimos os requisitos para esta etapa em três partes:

2.1 Verificação de Escopo

Doxa adota escopo estático como C ou Java, mas com algumas características próprias. Seguem as regras de escopo de Doxa:

1) Há **dois níveis de escopo** apenas: *global* e *local*. O nível local se refere ao corpo de uma função. Uma variável declarada localmente permanece ativa desde o ponto de sua declaração até o fim do corpo da função. (Isso também quer dizer que, se ela tiver sido declarada em um bloco, ela é visível fora).

2) Não pode haver mais de uma declaração do mesmo nome (identificador) no mesmo *nível de escopo*. Se houver, o compilador deve indicar um erro.
   - Observação: Para esta verificação, tratar nomes de variáveis e de funções indiscriminadamente. Assim, a declaração de uma variável **x** global vai conflitar com uma função **x()** (também global).

3) Nomes globais podem ser repetidos em uma declaração local. Neste caso, o nome local *esconde* o nome global (*name hiding*), enquanto ativo (ou seja, até o fim do corpo da função).

4) Toda variável e toda função precisam ser declaradas antes de usadas. Se uma variável ou função for usada sem ter sido declarada ou se for usada antes do trecho de código onde está sua declaração, o compilador deve indicar um erro.

No projeto, o tratamento dos nomes e dos seus escopos deve ser feito por meio de uma **tabela de símbolos**, construída conforme explicado nas aulas teóricas. Não usar tabela de símbolos ou implementá-la de forma muito diferente e ineficiente resultará em penalidade na nota.

2.2 Verificações de Tipos em Expressões

O compilador de **Doxa** deve analisar cada expressão, decompondo-a em subexpressões (se houver), para:

- **inferir o tipo** da expressão como um todo (que pode depender das subexpressões);

- **identificar e informar o erro principal** ao usuário (programador de Doxa), onde o **erro principal** é o erro presente na menor sub-expressão mais à esquerda[2].

Como você deve saber, a linguagem suporta os tipos numéricos **inteiro** (*int*) e **ponto flutuante** (*float*), além do tipo **caractere** (*char*). Adicionalmente, o compilador deve representar um tipo especial **booleano**, interno, que inclui apenas os valores *true* e *false*. Esse tipo é usado apenas internamente pelo compilador (i.e. programadores de **Doxa** não podem declarar variáveis desse tipo).

As regras de tipo de **Doxa** são:

1) As **operações aritméticas** binárias +, -, * e / aceitam operandos numéricos (int ou float) de mesmo tipo. Se ambos os operandos forem inteiros (int), a expressão resultante deve ser considerada do tipo inteiro. Se ambos forem *floats*, o a expressão é considerada do tipo *float*.

2) As operações aritméticas binárias +, -, * e / aceitam, também, operações entre um operando *float* e um *int*, nesta ordem. O resultado é *float*, nestes casos.

3) A operação de **resto da divisão** % só aceita dois operandos inteiros.

4) Também é permitida a **adição** (+) e a **subtração** (-) entre um caractere e um inteiro, nesta ordem. O resultado é do tipo *char*, nestes casos.

5) A **menos unário** "-" se aplica apenas a um operando numérico (*int* e *float*). Seu resultado é do mesmo tipo que seu operando.

6) Nas expressões com operadores aritméticos que apresentam qualquer outra combinação de operandos que não as previstas acima (nas regras 1 e 2), o compilador deve indicar erro.

7) As operações de **comparação <**, **<=**, **>**, **>=** só se aplicam a operandos numéricos de mesmo tipo (*int* com *int* ou *float* com *float*), e seu resultado é *booleano*. Para qualquer outra combinação de tipos, o compilador deve indicar erro.

8) As operações de **comparação ==** e **!=** se aplicam a todos os tipos. Ambos os lados devem ser do mesmo tipo. O resultado é *booleano*.

9) As **operações lógicas "and"** e **"or" e "not"** só se aplicam a operandos *booleanos*, resultando em um valor de tipo *booleano*.

10) A maior parte das expressões básicas tem tipos facilmente identificáveis. Porém, se for um identificador, o tipo é o da declaração de variável que está *ativa* para aquele identificador naquele ponto do código. Use a tabela de símbolos

[2] Ou seja: (i) se houver erros nos dois lados, mostrar apenas o da sub-expressão da esquerda; (ii) a mensagem de erro não deve ser propagada para as expressões que contêm a sub-expressão errada.

para identificar isso. Se for uma chamada de função, o tipo dela será o tipo de retorno da declaração da função. Use a tabela de símbolos também.

11) Os **tipos dos parâmetros reais** das funções (que são as expressões passadas na chamada da função) devem ser exatamente os mesmos dos **parâmetros formais** (as variáveis que aparecem como parâmetros na declaração da função) e devem vir na mesma ordem.

2.3 Verificações de Tipos em Comandos

Para os comandos que usam expressões, o compilador apenas irá verificar se o tipo delas é adequado. Se não for, o compilador deve indicar erro.

1) **Atribuição**: o nome do lado esquerdo deve ser uma variável (não pode ser função), e deve ter o mesmo tipo que a expressão do lado direito.

2) **Comandos de decisão e de iteração**. As expressões de teste presentes nestes comandos devem ser do tipo *booleano*.

3) **Comando de retorno**. A expressão em um comando "return" deve ter tipo igual ao tipo de retorno informado na declaração da função.

2.4 Outros Requisitos

A Análise Semântica deve parar no 1º erro identificado e **emitir uma mensagem clara de erro**. Evite fazer com que a execução principal do seu compilador seja abortada com uma exceção. (Você pode usar exceção internamente, desde que ela não “escape” para o usuário do compilador).

É recomendado (mas opcional) que indique linha e coluna de cada erro.

Um requisito extra é fazer a análise semântica detectar múltiplos erros. Ela não pararia no primeiro erro, mas continuaria a analisar o restante do código. Isso pode ajudar o programador usuário da linguagem a consertar mais de um problema por vez.

2.5 Pontuação

A Análise Semântica valerá **25%** da 2ª VA.

3. Geração de Código

Você tem duas opções, pelo menos: *gerar códigos de três endereços* ou *gerar código final* diretamente. Basta escolher uma delas, mas a escolha afeta a forma de calcular a nota. Ver as subseções a seguir.

3.1 Código Intermediário

Neste caso, seu compilador vai gerar um arquivo de texto de saída com um **código de três endereços** semanticamente equivalente ao programa Doxa dado como entrada.

O projeto vale até 70% da nota da 2ª VA, neste caso. Os outros 30% serão dados pela nota de uma prova.

3.2 Código Final

Neste caso, seu compilador vai gerar um arquivo de texto de saída com algum **código "assembly"** (para qualquer arquitetura) que seja semanticamente equivalente ao programa Doxa dado como entrada.

Não é necessário gerar código para *chamadas* de funções. É recomendado gerar código para **JVM**, usando a ferramenta **Jasmin** que define um "assembly extra-oficial" para a JVM, mas existem outras opções (e.g. Apache BCEL).

O projeto sozinho vale até 100% da nota da 2ª VA, neste caso. (O aluno ou dupla fica dispensado da prova).

3.3 Outros Requisitos

Abaixo, estão listados os principais requisitos gerais, válidos para qualquer das opções acima:

1) O seu gerador de código deverá produzir um código semanticamente equivalente ao código Doxa dado como entrada.

2) A **semântica** (significado) de cada comando de Doxa (atribuições, if-else, while, etc) é similar às dos comandos similares presentes nas linguagens C e Java. Qualquer dúvida no entendimento de algum deles, entrem em contato comigo.

3) Ao gerar código para avaliar as expressões é obrigatório **obedecer as precedências** definidas na primeira parte do projeto. (O parser oferecido no *projeto base* já garante esta ordem).

3.4 Pontuação

A pontuação da Geração de Código tem duas possibilidades:

- Se gerar código intermediário apenas, valerá **20%** da 2ª VA.
- Se gerar código final, independentemente de gerar o intermediário (que passa a ser dispensável), valerá **30%** da 2ª VA.