Compiladores
Trabalho Prático – Projeto de um Compilador

1 Introdução

A melhor forma de aprender sobre compiladores é construindo um! Esse é o objetivo do trabalho prático da disciplina: um projeto de desenvolvimento de um compilador, realizado ao longo de todo o curso, permitindo o estudo do conteúdo de forma prática.

O seu projeto deve ser realizado em grupo. Existem vários motivos para que o trabalho não seja individual, dentre os quais podemos citar:

- Um projeto em grupo facilita o desenvolvimento de um compilador para linguagens e arquiteturas reais, pois permite a distribuição das tarefas entre os integrantes do grupo.
- Um projeto em grupo permite simular o ambiente de desenvolvimento de *software* na sua futura vida profissional; praticamente todos os projetos são desenvolvidos por equipes de programadores que precisam coordenar e organizar o seu trabalho.

2 Formato da disciplina

A disciplina de Compiladores foi projetada para seguir um modelo de PBL (*Project Based Learning* – Aprendizado Baseado em Projeto). Sendo assim, o desenvolvimento do compilador é um elemento fundamental da disciplina, tanto para o aprendizado quanto para a avaliação.

Em uma disciplina PBL, o aluno deve tomar um papel mais ativo nos estudos, tentando buscar de forma independente as informações que necessitar. Nesse cenário, o papel do professor passa a ser o de curador de informações e mediador de dúvidas/problemas.

3 Interação com o professor

Vamos agora explicar a forma de funcionamento esperada da disciplina. Um ponto fundamental a se destacar é que é praticamente *impossível* desenvolver todo o projeto "de véspera" (isto é, deixando para fazer tudo próximo da data de entrega), porque o escopo do trabalho é grande. Como dito anteriormente, o projeto deve ser realizado *ao longo* curso, e não ao final. Assim, a forma de trabalho ideal é um desenvolvimento incremental ao longo de todas as semanas, contando com a participação de todos os integrantes do grupo.

É claro que não é esperado que os alunos façam tudo sozinhos, muito pelo contrário! Mais do que nunca, em uma disciplina PBL a interação dos alunos com o professor é essencial. Existem várias formas de interação possíveis, como a sala da disciplina no Classroom e no Google Chat, e-mail, e as aulas presenciais no laboratório. Espera-se que todos os alunos da disciplina participem e interajam ativamente entre si e com o professor. Via de regra, se você passou mais de uma semana sem conversar com o professor sobre o seu projeto, algo está errado...

4 Avaliação

Dado que não há provas parciais na disciplina, a avaliação do projeto do compilador determina de forma completa a sua média parcial no curso. A avaliação tem dois objetivos fundamentais:

1. Determinar o aprendizado do conteúdo por meio do código implementado.

2. Medir o nível de participação dos integrantes do grupo no desenvolvimento do projeto.

Para o item 1 acima, o professor vai analisar a versão final do código do compilador entregue no término do curso. (Mais detalhes do que é avaliado nesse ponto serão dados adiante.) Já para o item 2, serão realizados *checkpoints* (CPs – pontos de controle), onde serão determinados marcos específicos que o andamento do projeto deve atender. A quantidade, formato e pontuação dos CPs, bem como as informações sobre cálculo da média estão disponíveis no programa da disciplina. Fique particularmente atento às datas dos CPs, informadas no calendário da disciplina.

5 Etapas de desenvolvimento do projeto

Vamos agora discutir detalhadamente as atividades que você deve realizar para concluir com sucesso o projeto do seu compilador.

5.0 Atividade 0 – Começando

PRÉ-REQUISITOS (o que você já deve ter feito antes de começar essa atividade)**:**

- Ter estudado o conteúdo do Módulo 00.

PASSOS (o que você deve realizar nessa atividade)**:**

1. **Formar um grupo.** Você deve se juntar com os colegas da turma e formar um grupo. O tamanho ideal do grupo é de **três** integrantes. Escolha um **nome** legal para o grupo.
   Observação:
   - Nos casos em que o tamanho da turma não seja um múltiplo de 3, o professor vai mediar a criação de um ou dois grupos com **dois** integrantes, ajustando a dificuldade do projeto adequadamente para esses grupos menores. Não serão permitidos grupos com menos de 2 ou mais de 3 integrantes.
2. **Definir o escopo do projeto.** O seu grupo deve decidir qual **linguagem fonte** e qual **arquitetura alvo** o seu compilador vai tratar.
   Observações:
   - As opções para a escolha de uma linguagem de programação existente como a linguagem fonte a ser tratada vão ser disponibilizadas pelo professor. Em geral estas escolhas vão abranger LPs imperativas e estaticamente tipadas pois este é o estilo clássico de LP utilizada nos roteiros de laboratório. No entanto, você também pode propor uma LP de sua escolha, não sendo necessário focar somente no paradigma imperativo; de fato, alguns grupos acabam escolhendo linguagens mais "obscuras", como Haskell ou Lua. Não há relação direta entre a linguagem escolhida e a dificuldade do projeto: nos casos de linguagens com muitas características (*features*), o professor vai limitar adequadamente o escopo do projeto para manter o equilíbrio de trabalho entre os grupos. No caso do grupo propor uma LP, procure selecionar uma linguagem que você domina bem ou que queira aprender em mais detalhes.
   - As mesmas observações acima também valem para a escolha da arquitetura alvo de saída do seu compilador. Escolhas usuais incluem: MIPS, ARM, JVM e LLVM. Da mesma forma que no item anterior, variações menos usuais incluem x86 ou SPARC, aonde a escolha depende da afinidade e interesse de cada grupo. Note que não há problema em se utilizar uma arquitetura virtualizada, como por exemplo a JVM; essa é na verdade uma característica muito comum de linguagens mais recentes. Nos casos

de compilação para arquiteturas não-virtualizadas, o projeto termina na geração de código *assembly*; não é necessário implementar a parte de montagem e ligação pois vamos usar os *assemblers* e *linkers* já existentes para a arquitetura.
    - Uma linguagem usual para *implementação* do seu compilador é C, e o material da disciplina (incluindo todos os laboratórios) foi desenvolvido com isso em mente. No entanto, isso não impede que você realize o seu trabalho em outra linguagem. Uma segunda escolha usual é Java, por conta da ferramenta ANTLR para geração de analisadores léxicos e sintáticos, que é muito utilizada atualmente.
3. **Recolher documentação e ferramentas.** Uma vez definido o escopo do seu projeto, você deve começar a reunir as informações necessárias para a sua execução. Um ponto de partida essencial é o manual da linguagem fonte escolhida, que define as características léxicas, sintáticas e semânticas da linguagem. Da mesma forma, uma documentação sobre a arquitetura alvo também será necessária no futuro. Note que se você pretende gerar código para uma arquitetura não-virtualizada à qual você não possui acesso, será necessário também um simulador (ex. para ARM, MIPS, etc) para ser possível testar o código gerado pelo seu compilador.
4. **Montar um ambiente para o trabalho.** Procure já ir construindo um ambiente colaborativo de desenvolvimento com o seu grupo. Por exemplo, forme grupos de comunicação online entre os integrantes. Além disso, o uso de um repositório de código (ex. `git`) é extremamente recomendável.

RESULTADOS (o que você deve fazer para concluir essa atividade)**:**

- Você deve registrar com o professor todas as informações sobre o seu grupo (nome, integrantes, linguagens fonte e alvo) até as datas limites.

5.1 Atividade 1 – Analisador Léxico

PRÉ-REQUISITOS:

- Ter estudado o conteúdo do Módulo 01.
- Ter concluído com sucesso a tarefa do Laboratório 01.

PASSOS:

1. **Identificar os elementos léxicos da linguagem fonte.** Você precisa conhecer todos os *tokens* que existem na linguagem, bem como entender como deve ser o tratamento de comentários e espaços em branco (tabulações, enter, etc). Em geral, todas essas informações estão presentes no manual da linguagem.
2. **Implementar o *scanner*.** Baseado nas informações do item anterior, implemente o *scanner* do seu compilador usando o `flex`, o ANTLR, ou outra ferramenta de sua preferência. Em geral esse passo é razoavelmente simples. Normalmente é possível tratar todos os elementos da linguagem sem nenhuma simplificação. Em Python, e outras LPs nas quais espaçamento possui significado sintático, geralmente a análise léxica é um pouco mais complexa por conta da definição de blocos de código.
3. **Desenvolver casos de teste.** Crie uma série de casos de teste (programas de entrada) simples que permitam testar todos os aspectos da implementação do seu *scanner*. Baseie-se nos exemplos utilizados na linguagem do laboratório para ter uma ideia do que fazer. Algumas linguagens possuem suítes de testes específicas para testar os seus compiladores (ex. Lua); você é livre para usar esses testes se quiser.

RESULTADOS:

- Esta atividade é a primeira parte do primeiro *checkpoint* (CP1). Para a entrega de CP1 você deve produzir um breve relatório, aonde detalhes da implementação relevantes devem ser destacados (ex. como foram tratados comentários, indentação em Python, etc). Também devem ser discutidos os principais casos de teste, bem como a justificativa para a sua escolha.

5.2 Atividade 2 – Analisador Sintático

PRÉ-REQUISITOS:

- Ter estudado o conteúdo do Módulo 02.
- Ter concluído com sucesso a tarefa do Laboratório 02.

PASSOS:

1. **Identificar os elementos sintáticos da linguagem fonte.** Você deve conhecer toda a sintaxe da linguagem fonte. Em geral, essas informações estão presentes no manual da linguagem, aonde é bastante comum a inclusão da gramática geradora da linguagem em formato BNF (ou EBNF).
2. **Implementar o *parser*.** Baseado nas informações do item anterior, implemente o *parser* do seu compilador usando o `bison`, o ANTLR, ou outra ferramenta de sua preferência. (Obviamente, você deve incorporar também o *scanner* da atividade anterior.) É bastante provável que a transcrição da gramática do manual para o `bison` leve a conflitos de *shift/reduce* ou de *reduce/reduce*. Esses conflitos são causados por ambiguidades na gramática. Como nenhuma linguagem de programação pode ser ambígua, a definição da linguagem (manual) certamente estabelece critérios para desambiguação, como por exemplo precedência de operadores, etc. **Todos** os conflitos devem ser resolvidos nesse momento pois eles certamente vão causar erros nas próximas atividades. Alguns problemas são facilmente resolvíveis, já outros (normalmente conflitos de *reduce/reduce*) requerem adaptações da gramática. Nessa atividade normalmente ainda é possível tratar (quase) toda a gramática da linguagem, com nenhuma ou poucas simplificações.
3. **Testar o *parser*.** Você pode usar os casos de teste desenvolvidos na atividade anterior, mas talvez também seja necessário criar mais alguns. Certifique-se de exercitar todos os possíveis construtos da linguagem (*loops*, testes, expressões, etc) para encontrar possíveis erros de implementação da gramática; um analisador sintático funcional é essencial para as próximas atividades.

RESULTADOS:

- Esta atividade é a segunda parte do primeiro *checkpoint* (CP1). Inclua no relatório os detalhes de implementação relevantes para o *parser* (ex. remoção de conflitos na gramática). Discuta também os casos adicionais que foram incluídos para testar o *parser*, se for o caso.

5.3 Atividade 3 – Analisador Semântico

PRÉ-REQUISITOS:

- Ter estudado o conteúdo dos Módulos 03 e 04.
- Ter concluído com sucesso as tarefas dos Laboratórios 03 e 04.

PASSOS:

1. **Determinar a semântica da linguagem fonte.** Nesse ponto a atividade fica um pouco mais complexa pois a semântica da linguagem geralmente está espalhada ao longo do manual, sendo descrita em várias páginas de texto. Se você já tem um bom domínio sobre a linguagem sendo tratada, provavelmente uma breve leitura do manual deve ser suficiente. Caso contrário, será necessário estudar adequadamente essas informações antes de partir para a implementação.

2. **Implementar o analisador semântico.** Baseado nas informações do item anterior, implemente o analisador semântico do seu compilador. Essa etapa de compilação normalmente contempla várias verificações, tais como checagem de tipos, etc.
   Observações:
   - Algumas linguagens dinâmicas requerem que pelo menos parte da análise semântica seja feita durante a execução do programa. Para poder realizar essa atividade, considere inicialmente que a linguagem é estática e procure fazer a maior quantidade possível de verificações semânticas durante a compilação (nem todas as verificações serão possíveis por conta da própria natureza das linguagens dinâmicas). Depois que essa análise estiver implementada, ela deve ser movida para outras fases do compilador nas atividades seguintes.
   - Nesse ponto é possível que linguagens monstruosas (ex. Java, C#) tenham a sua semântica simplificada para caber no escopo do projeto da disciplina. Essas simplificações sempre devem ser combinadas antes com o professor.

3. **Testar o analisador semântico.** Você pode usar os casos de teste desenvolvidos na atividade anterior, mas talvez também seja necessário criar mais alguns. Certifique-se de exercitar todos os testes que você implementou para encontrar possíveis erros no código do analisador.

RESULTADOS:

- Esta atividade é a primeira parte do segundo *checkpoint* (CP2). O relatório do CP2 deve incluir detalhes de implementação relevantes do analisador semântico (ex. como foi desenvolvido o sistema de tipos). Também devem ser discutidos os casos de teste, bem como a justificativa para a sua escolha.

5.4 Atividade 4 – Código Intermediário

PRÉ-REQUISITOS:

- Ter estudado o conteúdo do Módulo 05.
- Ter concluído com sucesso a tarefa do Laboratório 05.

PASSOS:

1. **Definir o formato de representação intermediária.** No laboratório foi utilizada a AST (*Abstract Syntax Tree*) como representação intermediária (*Intermediate Representation* – IR) do código de entrada, mas essa não é a única opção.
   Observações:
   - Existem várias IRs possíveis, tanto em formato ramificado (AST) quanto em formato linear (ex. RTL – *Register Transfer Language*, usado no `gcc`). Você é livre para escolher qualquer IR que quiser; não usar nenhuma também é uma opção... :-P

- Usar uma AST tem como vantagem prover uma representação mais estruturada do código, o que pode facilitar a etapa seguinte de geração.
- Por outro lado, ao escolher um formato linear, você já vai estar mais próximo do código real para a arquitetura alvo.
- Uma outra alternativa é pular totalmente essa fase e partir diretamente para a geração de código. Os compiladores antigos eram implementados assim, mas, além de dificultar bastante qualquer otimização no código, essa forma também complica mais o processo de geração de código alvo ao final.

2. **Implementar e testar a geração da IR.** Quando aplicável, desenvolver o processo de geração conforme definido no passo anterior e testar.

RESULTADOS:

- Esta atividade é a segunda parte do segundo *checkpoint* (CP2). Inclua no relatório os detalhes de implementação relevantes para a geração da IR.

5.5 Atividade 5 – Geração de Código

PRÉ-REQUISITOS:

- Ter estudado o conteúdo dos Módulos 06 e 07.
- Ter concluído com sucesso as tarefas dos Laboratórios 06 e 07.

PASSOS:

1. **Determinar um subconjunto de características da arquitetura alvo.** Via de regra, todas as arquiteturas existentes são bastante complexas e possuem várias instruções e modos de operação que não são necessários para esse projeto (ex. instruções SSE na arquitetura x86). Você deve estudar a documentação da arquitetura alvo e escolher o conjunto mais simples possível de instruções que serão utilizadas no *backend* do seu compilador.

2. **Implementar a geração de código alvo.** Dependendo da configuração da arquitetura alvo, pode ser necessário utilizar como estrutura interna do *backend* uma pilha (código de 1-endereço) ou uma lista de quádruplas (código de 3-endereços). Basei-se no que você aprendeu nos laboratórios indicados acima para tomar essa decisão.

3. **Testes finais do compilador.** Ao gerar código *assembly* correto ou, conforme o caso, outro código de uma arquitetura virtualizada, você deve ser capaz de enfim executar os seus casos de teste. Nesse estágio ainda pode ser necessário simplificar a linguagem tratada, principalmente no aspecto de uso da memória. Por exemplo, uma gerência elaborada de *heap* e coleção de lixo normalmente estão fora do escopo desse projeto.

RESULTADOS:

- O grupo deve apresentar o seu compilador completo para o professor e a turma através de um seminário (CP3). Todos os detalhes de implementação relevantes devem ser destacados, principalmente a parte de geração de código. Também devem ser exibidos os casos de teste, bem como a justificativa para a sua escolha.

6 Comentários finais

- A avaliação/correção da implementação do trabalho leva em conta os seguintes aspectos:

1. **Corretude**: um compilador que gera código errado obviamente é inútil. Assim, é de se esperar que o ponto fundamental na avaliação do seu compilador seja o seu correto funcionamento.
2. **Cobertura dos testes**: esse ponto está diretamente ligado ao anterior porque um bom conjunto de testes tem uma relação direta com o funcionamento adequado de um programa.
3. **Qualidade/organização do código**: assumindo um compilador funcional, o ponto seguinte na avaliação leva em consideração a qualidade do código desenvolvido no projeto. Existem inúmeras recomendações sobre o que distingue um código bom de um ruim (por exemplo, Clean Code); escolha uma que se adeque ao seu estilo de programação e use-a de forma consistente.
4. **Documentação**: o código do seu compilador deve estar bem comentado. Note que **bem** não é sinônimo de **muito**, pelo contrário! Citando novamente os princípios de Clean Code: *"Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments."* Via de regra, a sua intenção como programador deve estar diretamente expressa no código, deixando os comentários para esclarecimento dos pontos mais complexos da implementação. (Como exemplo, veja a seção 3 em `https://medium.com/better-programming/clean-code-5-essential-takeaways-2a0b17ccd05c`).

- A avaliação dos CPs vai se basear nos seguintes critérios:
  1. **Domínio do conteúdo**: todos os integrantes do grupo devem ter um completo entendimento de todas as partes do código do compilador, mesmo aquelas que não tenham sido diretamente escritas pela pessoa.
  2. **Participação no trabalho do grupo**: todos os integrantes do grupo devem ter uma participação razoavelmente igualitária no desenvolvimento do projeto. Alunos que não seguirem esse princípio vão prejudicar todo o grupo, porque a nota do CP é coletiva e não individual.
- Como mensagem final, o professor gostaria de destacar que essa disciplina é interessante por condensar conteúdos de várias outras matérias do curso. Por conta disso, esse projeto serve como um bom balizador das suas habilidades como alguém que vai formar em breve em um curso de Computação. O projeto foi pensado para ser uma atividade desafiadora mas sem ser extenuante. Espera-se que você ache esse trabalho interessante, e, quem sabe, talvez até divertido... Bom trabalho!