Projeto de Compiladores

1 de Março de 2019

Departamento de Engenharia Informática

1 - Generalidades

Este projeto pretende desenvolver um compilador para uma linguagem de programação. Este compilador deverá efetuar a análise sintática e semântica da linguagem, produzindo como resultado código máquina (*assembly*) com seleção de instruções em formato **nasm** para **linux-elf32-i386**. O compilador deverá ser escrito em **C/C++** com auxílio das ferramentas de análise lexical, sintática e seleção de instruções, respetivamente, flex, byacc e pburg. A geração automática de ficheiros intermédios deverá ser feita com recurso à ferramenta **gmake**, devendo o projeto incluir as *Makefile*s e todos os ficheiros necessários à sua construção.

A descrição da linguagem está disponível no manual de referência disponibilizado em separado. Esta descrição deve ser seguida rigorosamente, não sendo valorizada qualquer modificação ou extensão, antes pelo contrário.

O projeto é desenvolvido individualmente. Apenas se consideram para avaliação os projetos submetidos no Fenix, contendo todos os ficheiros fontes necessários à resolução do projeto, em formato .tgz (ou .zip), até à data limite. A submissão de um projeto pressupõe o compromisso de honra que o trabalho incluso foi realizado pelo aluno. A quebra deste compromisso, ou seja a tentativa de um aluno se apropriar de trabalho realizado por colegas, tem como consequência a reprovação de todos os alunos envolvidos (incluindo os que possibilitaram a ocorrência) à disciplina neste ano letivo.

Parte da nota do projeto é obtida através de testes automáticos, pelos que estas instruções devem ser seguidas rigorosamente. Caso os testes falhem por causas imputáveis ao aluno a nota refletirá apenas os testes bem sucedidos. As restantes componentes da nota são obtidas pela análise do código entregue e pela avaliação do teste prático. O código é avaliado quanto à sua correção, simplicidade e legibilidade. Para tal, os comentários, nomeadamente, não devem ser excessivos nem óbvios, por forma a dificultar a legibilidade, nem muito escassos, por forma a impedir a compreensão das partes mais complexas do programa.

1.1 - Formato dos ficheiros

O ambiente de trabalho da cadeira é **C/C++** sobre **Linux** em arquiteturas **i386** com formato **elf32**. Embora os alunos possam fazer grande parte do desenvolvimento noutras máquinas com outros sistemas operativos, uma vez que quase todas as ferramentas existem disponíveis em código fonte, a avaliação pressupõe o seu desenvolvimento em **linux-elf32-i386**. Logo, todos os nomes dos ficheiros são em minúsculas, salvo referência explícita em contrário. As linhas dos ficheiros terminam exclusivamente em carriage-return (mudança do carreto, em português), ou seja o código 0x0A (10 em decimal ou 012 em octal). Quaisquer erros resultantes da existência de outros carateres, ou comportamentos específicos, são tratados como qualquer outro erro. Exemplos de teste que produzam erros

na execução dos testes serão ignorados na avaliação.

1.2 - Dúvidas

Nos casos omissos e se surgirem dúvidas na interpretação da especificação da linguagem de programação deve primeiro consultar a errata na secção de projeto disponível na página da cadeira e só depois, caso a dúvida subsista, consulte o professor responsável da cadeira.

2 - Programas em diy

Antes de iniciar o desenvolvimento do compilador, propriamente dito, deverá desenvolver alguns programas de exemplo de aplicação da linguagem. Pretende-se, desta forma, verificar a correta compreensão da linguagem de programação **diy**.
A escrita dos programas obriga a uma cuidadosa leitura do manual de referência por forma a garantir que os programas escritos não irão gerar erros de compilação ou execução. Estes mesmos programas servirão como padrões de teste quando o aluno estiver a desenvolver o compilador para a linguagem **diy** . Na entrega os programas deverão ter os nomes indicados. Os ficheiros a serem gerados pelo compilador têm o mesmo nome dos ficheiros fonte, mas com a extensão .asm em vez de .diy.

2.1 - Módulo de manipulação de cadeias de carateres

O módulo será designado string.diy (e não String.diy, STRING.DIY, STRING.diy ou outra variante que a imaginação considere mais apropriada) e deverá conter os equivalentes em **diy** das rotinas de **C**: strcmp, strcpy e strchr. (Como não existem carateres individuais na linguagem, o segundo argumento da rotina strchr é um integer) A rotina strcmp deverá ser utilizada pelo compilador final para a comparação de cadeias de carateres.

2.2 - Programa de cálculo do fatorial

O programa será constituído por 2 módulos e um programa,

designados respetivamente por iter.diy, recurs.diy e factorial.diy. Os 2 módulos definem apenas a função factorial, que recebe um argumento inteiro e devolve um número real em vírgula flutuante, que calcula o diy do argumento iterativamente e recursivamente, respetivamente. O programa recebe como argumento opcional da linha de comandos o valor do diy a calcular e imprimir, devendo calcular o fatorial de **5** caso o argumento não seja indicado. Como as duas funções têm o mesmo nome apenas uma pode ser usada na produção do ficheiro executável final. Assim, a Makefile, em ambas as entregas, deverá produzir 2 ficheiros executáveis, iter e recurs, ambos usando a mesma rotina principal e cada um usando a respetiva versão do cálculo do diy. Não deve utilizar o operador fatorial (!) uma vez que será um destes módulos a fornecer a rotina que realiza o referido operador quando o compilador estiver concluído.

2.3 - Média de sequências pseudo-aleatórias

O programa será constituído por um módulo e um programa, designados respetivamente por rand.diy e mean.diy. O módulo realiza uma rotina (public integer rand()) de geração de números inteiros pseudo-aleatórios, segundo o algoritmo congruencial com multiplicador 27983, incremento 149 e módulo 1000000. Ou seja, X_{n+1} = (X_n * mul + incr) % mod. O valor inicial, não negativo e inferior ao módulo, é iniciado a X_0 = 100003, podendo ser alterado pela rotina public void srand(integer), que também faz parte do módulo. O programa lê do terminal o número de valores pseudo-aleatórios a gerar e imprime a média em vírgula flutuante dos valores gerados.

3 - Compilador de diy

É fortemente aconselhado proceder da seguinte forma na realização do projeto:

- Estruture previamente o compilador, identificando os seus módulos

principais.
- Especifique com particular cuidado as estruturas de dados a usar. Caso opte por usar as versões de código disponibilizado pela cadeira, deve compreender o seu funcionamento.
- Note que é mais fácil a modularização de um programa recorrendo a funções curtas com a funcionalidade limitada e bem definida.
- Não tente definir a linguagem completa de uma só vez, começando por uma ou duas instruções (por exemplo, escrita e atribuição), acrescentando gradualmente as restantes instruções e operadores.
- Desenvolva cada uma das fase de análise e geração sucessivamente, não iniciando a seguinte antes de eliminar os erros das anteriores. Utilize diversos ficheiros de exemplo, incluindo os exemplos distribuídos, para testar as funcionalidades do compilador nas suas sucessivas fases.

O desenvolvimento do compilador, em especial a geração de código, deve evoluir de forma a fazer programas em que o resultado é observável. Assim, sugere-se a seguinte sequência:

- Invocação de rotinas, em especial a impressão de cadeias de carateres.
- Apenas definições globais, impressão e expressões do tipo inteiro.
- Instruções condicionais e de ciclo.
- Definição de funções.
- Tipo number para números reais.
- Expressões lógicas e tipo referência.
- Outras declarações e restante linguagem.

Para auxiliar o desenvolvimento do compilador é distribuído um compilador, designado por **compact** para uma linguagem simples, podendo os ficheiros distribuídos ser modificados sem restrições. Este compilador encontra-se na página da disciplina do Fénix.

3.1 - Análise lexical

A análise lexical da linguagem **diy** deverá ser realizada com o auxílio da ferramenta **flex** (invocada obrigatoriamente com a opção **-l**). Com a análise lexical deverá ser possível remover comentários e espaços brancos, identificar literais (valores constantes), identificadores (nomes de variáveis e funções), palavras chave, etc. Notar que a análise lexical não garante que estes elementos se encontrem pela ordem correta.

O ficheiro da descrição lexical deverá ser processado pela ferramenta flex sem gerar qualquer tipo de avisos. As sequências de escape nas cadeias de carateres deverão ser substituídas pelos respetivos carateres, o espaço necessário para o texto dos identificadores e cadeias de carateres deverá ser reservado antes de devolvido. Os valores literais deverão ser convertidos para o respetivo formato binário e testada a capacidade de representação da máquina. Na avaliação serão analisadas as expressões regulares utilizadas, bem como a robustez, clareza, simplicidade e extensibilidade da solução proposta.

Deverá ser produzido um ficheiro designado **diy.l**, contendo a análise lexical tal como será utilizada no compilador final.

3.2 Análise sintática

A análise sintática da linguagem **diy** deverá ser realizada com o auxílio da ferramenta **byacc**. Com a análise semântica deverá ser possível garantir a correta sequência dos símbolos, embora não se verifique se as variáveis utilizadas nas expressões estão declaradas, se algumas das operações suportam os tipos de dados utilizados, etc.

O ficheiro da gramática deverá ser processado pela ferramenta byacc sem gerar qualquer tipo de conflitos (shift-reduce ou reduce-reduce) nem erros ou avisos de qualquer espécie. Caso o ficheiro a processar contenha erros, estes devem ser identificados e descritos ao utilizador. Além disso, mesmo que sejam encontrados erros, o ficheiro em análise deve continuar a ser processado até ao fim, procurando outros erros.

Na avaliação será analisada a gramática entregue do ponto de vista verificações de consistência tratadas sintaticamente, das regras escolhidas e símbolos não terminais escolhidos, bem como a robustez, clareza, simplicidade e extensibilidade da solução proposta.

Posteriormente, podem ser associadas ações que permitam construir uma árvore de análise sintática do programa a ser processado.

Deverá ser produzido um ficheiro designado **diy.y**, contendo a análise sintática tal como será utilizada no compilador final.

3.3 - Análise semântica

A análise semântica da linguagem **diy** deverá garantir que um programa descrito na linguagem **diy** se encontra corretamente escrito e que pode ser executado.

Como resultado da análise semântica deverão ser identificados todos os erros estáticos (detetáveis no processo de compilação) ainda não detetados nas fases anteriores e produzidas mensagens de erro descritivas. Caso surjam erros semânticos (estáticos), o ficheiro deve ser integralmente processado e o compilador deverá terminar com um código de erro 1 (um). A deteção de qualquer na fase de análise inibe a geração de código, não devendo ser gerado qualquer ficheiro.

A análise semântica deve verificar, entre outros, a prévia declaração dos identificadores a utilizar, bem como a consistência de tipos das operações permitidas pela linguagem (tabela de símbolos). Notar que após a análise semântica não podem ser identificados mais erros no programa fonte a compilar. Logo nenhuma verificação necessária pode ficar por efetuar após a análise semântica.

3.4 - Construção da árvore sintática

A construção da árvore sintática deve incluir toda a informação necessária à geração de código e ser efetuada de tal forma que possa ser processada pela ferramenta pburg. Para tal deve ter em consideração que a ferramenta apenas considera os dois primeiros ramos de cada nó da árvore. Além disso, os nós devem representar instruções tão próximas quanto possível das primitivas básicas de um processador genérico.

A geração da árvore sintática deverá ser incluída nas ações do ficheiro de análise sintática **diy.y**. Este ficheiro pode ser livremente alterado da entrega intermédia para a entrega final, incluindo as próprias regras gramaticais, tal como o ficheiro **diy.l** Nesta fase, e apenas para efeitos de *debug* interno, sugere-se que o compilador imprima árvore de análise sintática do programa **diy** lido, se o programa estiver lexical, sintática e semanticamente correto.

3.5 - Geração de código final otimizado

A geração de código final deverá ser realizada com o auxílio da ferramenta pburg. Para gerar código final aceitavelmente otimizado deve descrever gramaticalmente as capacidades do processador alvo em função da árvore sintática gerada num ficheiro de reescrita de árvores. Este ficheiro define a forma de acesso à árvore sintática, os símbolos terminais da árvore sintática e a gramática das instruções e respetivos custos. A seleção das instruções deve aproveitar da melhor forma possível as capacidades do processador. Notar que como a geração de código utiliza as macros **postfix**, a seleção de instruções deve aproveitar o melhor possível as capacidades das instruções disponíveis. Para o cálculo dos custos deve-se considerar que cada instrução **postfix** tem um custo unitário (1), excluindo as diretivas *assembly* que têm um custo nulo (0).

O ficheiro *assembly* gerado deve ter um formato que seja processado pelo *assembler* **nasm**, supondo o sistema operativo **Linux** e o formato de ficheiro **elf32**. Para tal, o comando **nasm -felf file.asm** permite gerar o

ficheiro file.o a partir do ficheiro file.asm, assumindo que não são encontrados erros no seu processamento.

Deverá ser produzido um ficheiro designado **diy.brg**, contendo a geração do código final tal como será utilizada no compilador final.

3.6 - Biblioteca de suporte à execução

Para poder executar os programas gerados, a biblioteca de suporte à execução deve fornecer todas as rotinas necessárias à execução das operações suportadas pela linguagem. A biblioteca, designada por libdiy.a, deve também incluir o código necessário ao arranque do programa e invocação da rotina inicial entry.

Para tal, o comando **ld -o programa file.o libdiy.a** permite gerar o ficheiro programa a partir do ficheiro file.o, assumindo que não são encontrados erros no seu processamento.

4 - Avaliação

A avaliação do projeto é constituída por:

1. resultados produzidos pelos programas de exemplo de aplicação e pelo compilador da linguagem **diy** face a um conjunto de padrões de teste.
2. análise do código entregue, do ponto de vista da robustez, clareza, simplicidade e extensibilidade.

NOTA: qualquer alteração à especificação é penalizada, mesmo que possa ser entendida como um melhoramento. Pretende-se realizar um compilador para a linguagem **diy** e não que invente uma nova linguagem, mesmo que baseada em **diy**.

Notar que o facto de os testes terem sido superados não reflete a qualidade

do código, quer do ponto de vista de engenharia de software, quer do ponto de vista da correta aplicação dos princípios lecionados nesta disciplina. A funcionalidade do compilador final é de extrema importância, pelo que é preferível um programa que realize, corretamente, apenas parte da funcionalidade face a um quase completo mas que nem sequer compila ou que não gera nenhum programa correto.

4.1 - Entrega inicial

Entregar exemplos descritos na secção 2, o ficheiro de especificação **lex** (diy.l) e um ficheiro **yacc** (inicial.y) mínimo que permita a impressão dos *tokens* processados pelo analisador lexical (ver último exercício do laboratório 3), além de uma Makefile que produza o executável diy.

Para a avaliação inicial considera-se a última versão submetida ao fénix anterior às 17 horas do dia 22 de Março de 2019 (hora do fénix). Esta avaliação contribui com **6** valores para a nota do projeto.

4.2 - Entrega intermédia

Entregar os ficheiros de especificação **lex** (diy.l) e **yacc** (diy.y) que realizem a análise semântica e, caso o programa esteja correto, imprime a árvore sintática abstrata (**AST**). Inclua igualmente uma Makefile que produza o executável diy e todos os ficheiros auxiliares que forem necessários.

A execução do referido programa deverá devolver 0 (zero) se o programa não tiver erros lexicais, sintáticos ou semânticos. O programa deverá devolver 1 (um) e produzir mensagens de erro esclarecedoras no terminal (usando o stderr) caso o programa tenha erros lexicais, sintáticos ou semânticos.

Para a avaliação intermédia considera-se a última versão submetida ao fénix anterior às 17 horas do dia 12 de Abril de 2019 (hora do fénix). Esta

avaliação contribui com **6** valores para a nota do projeto.

4.3 - Entrega final

Entregar os ficheiros de especificação **lex** (diy.l), **yacc** (diy.y) e **burg** (diy.brg) que produza, caso o programa esteja correto, um ficheiro *assembly* **i386** com o mesmo nome do ficheiro de entrada e a extensão .asm. Inclua igualmente uma Makefile que produza o executável diy e a biblioteca de execução libdiy.a, bem como todos os ficheiros auxiliares que forem necessários.

A execução do referido programa deverá devolver 0 (zero) se o programa não tiver erros lexicais, sintáticos ou semânticos. O programa deverá devolver 1 (um) e produzir mensagens de erro esclarecedoras no terminal (usando o stderr) caso o programa tenha erros lexicais, sintáticos ou semânticos.

Para a avaliação final considera-se a última versão submetida ao fénix anterior às 17 horas do dia 24 de Maio de 2019 (hora do fénix). Esta avaliação contribui com **8** valores para a nota do projeto.

4.4 - Teste prático

O teste prático tem por objetivo garantir o domínio do projeto entregue por parte do aluno. O teste prático é individual. O teste prático consiste na realização de pequenas alterações ao projeto submetido, bem como à verificação da sua correta aplicação a exemplos de teste.

O teste prático tem a duração máxima de 1 (uma) hora. O teste prático realiza-se no dia 27 de Maio de 2019 entre as 9h e as 20h (e caso seja necessário, nos dias seguintes). Para ter acesso ao teste prático o aluno deve ter pelo menos **5.7** valores na soma das 3 entregas, caso contrário é impossível obter a nota mínima de **9.5** valores.

(C)IST, *Pedro Reis dos Santos*, 2019-02-18