SCC: Um Compilador C como Ferramenta de Ensino de Compiladores

Juliano Henrique Foleiss, Guilherme Puglia Assunção, Eduardo Henrique Molina da Cruz, Ronaldo Augusto de Lara Gonçalves, Valéria Delisandra Feltrim
Universidade Estadual de Maringá / Departamento de Informática
{juliano.foleiss, guilherme.assuncao, eduardo.cruz, ronaldo, valeria.feltrim}@din.uem.br

Resumo

O SCC (SOIS C Compiler) é um compilador da linguagem C padrão ANSI que gera código de montagem compatível com o SASM, o montador do ambiente SOIS. O SCC foi desenvolvido para permitir que programas possam ser escritos em C e executados sob inspeção. O compilador SCC oferece recursos que facilitam o ensino e a aprendizagem de compiladores incluindo modos de depuração passo a passo e visualização dos artefatos gerados durante o processo de compilação. Experimentos com o simulador mostram a sua aplicabilidade e importância.

1. Introdução

O SOIS (Sistema Operacional Integrado Simulado) [5, 9] é um ambiente de simulação integrado, envolvendo a simulação parametrizada de Compilador, Sistema Operacional, Processador e Sistema de E/S, possibilitando a escrita, execução e depuração de programas reais em um ambiente simulado.

Em uma etapa anterior [8], o montador SASM foi desenvolvido, que por sua vez lê um programa escrito em linguagem de montagem e gera um arquivo com código objeto da arquitetura IA-32 [10]. O intuito era facilitar o desenvolvimento do compilador, deixando a cargo deste a tradução da linguagem C para um código de montagem compatível com o SASM.

Embora existam diversos compiladores da linguagem C para a arquitetura IA-32, sendo o exemplo mais conhecido o compilador GCC da GNU [12], justifica-se o desenvolvimento de um compilador da linguagem C para o SOIS pelo fato do simulador não contemplar todo o conjunto de instruções. Além disso, este trabalho também tem como objetivo proporcionar facilidades que ajudem no ensino de disciplinas que abordem o processo de compilação de linguagens imperativas de alto nível.

A entrada do compilador é um programa escrito em linguagem C e sua saída é uma listagem de código de montagem compatível com o montador SASM. O programa também pode ser automaticamente montado, sem a necessidade de invocar o montador manualmente.

O SCC é capaz de gerar código para todas as construções da linguagem C, tais como tipos de dados definidos por usuário, ponteiros, todas as instruções de controle e repetição, enumerações, atribuição de valores iniciais às variáveis e outras características comuns as linguagens imperativas.

Modos de depuração foram implementados, de maneira que todo o processo de compilação pode ser detalhado passo-a-passo. Assim como no SASM, que possui funcionalidade similar, os modos de depuração podem ser utilizados no ensino, com o intuito de mostrar todas as fases do processo de compilação, a inter-relação de seus componentes e artefatos, além de permitir a visualização dos resultados gerados.

A Seção 2 deste artigo apresenta alguns trabalhos relacionados. Na Seção 3 é feita uma revisão do processo de compilação. A Seção 4 expõe brevemente o precursor do SCC, o SASM. A Seção 5 explica o SCC em termos de sua implementação, estruturas de dados e algoritmos. A Seção 6 mostra alguns casos em que o SCC pode ser utilizado para o auxílio no ensino de disciplinas que abordem o processo de compilação. Finalmente, na Seção 7, são apresentadas as conclusões e possíveis trabalhos futuros.

2. Trabalhos relacionados

Poucos trabalhos têm sido desenvolvidos com o propósito de ensinar o processo de compilação, conforme relatados nesta seção. O compilador educacional Verto [11] foi escrito na linguagem de programação Java. A linguagem compilada é baseada em português estruturado. O foco principal do Verto são as fases finais do processo de compilação, síntese

de código intermediário e geração de código-objeto, utilizando uma técnica de análise léxica simples e um método de análise sintática recursiva descendente. O aluno tem a oportunidade de consultar a saída de cada uma das fases da compilação por meio de uma interface de janelas, podendo visualizar as saídas do analisador léxico, a tabela de símbolos, a seqüência de regras sintáticas e as movimentações na pilha semântica. A principal deficiência desse compilador é que a linguagem alvo é baseada em um conjunto bem reduzido de instruções, o qual é executado pelo simulador arquitetural hipotético Cesar.

Outro compilador educacional, o ComPas [6], tem como objetivo principal ensinar como o compilador trabalha, mostrando por meio de tabelas comentadas as informações geradas durante o processo de compilação. O compilador pode gerar código Intel ou código para um processador de propósitos educacionais E97 a partir de programa fonte escrito em um subconjunto da linguagem Pascal. O compilador permite demonstrar diversas características de alto nível, incluindo, variáveis, tipos de dados, organização dos dados na memória RAM, métodos das principais estruturas algorítmicas e detalhes do uso de procedimentos, entre outras.

A ferramenta C-Gen [2] objetiva possibilitar a geração de compiladores por parte do usuário, exibindo o funcionamento do processo de reconhecimento passo a passo por meio de uma interface gráfica. A ferramenta permite o uso de diferentes métodos de reconhecimento para cada passo da compilação. Os passos da compilação são implementados como *plugins* que devem reconhecer as respectivas entradas e produzir saídas correspondentes. Existem *plugins* para a edição de autômatos finitos, geração de analisadores sintáticos e criação de analisadores semânticos.

Em [4] é apresentado o simulador CompilerSim, desenvolvido em Delphi, com objetivo de permitir a experimentação de conceitos e princípios das fases de um compilador em salas de aula, como ferramenta de apoio as disciplinas que abordam as áreas de compiladores. O simulador executa um processo de análise de código de programação baseado em subconjunto da linguagem Pascal, não tratando aspectos envolvendo registros, vetores, operações de E/S em arquivos, ponto-flutuante e otimização de código, entre outras simplificações. A interface gráfica se limita basicamente a mostrar os *tokens* identificados na análise léxica.

O compilador SCC proposto neste artigo difere dos trabalhos relatados nesta seção em quatro aspectos: (i) foca em especial o ensino da análise semântica e da geração de código, uma vez que entendemos serem essas as fases mais complexas no desenvolvimento de compiladores; (ii) gera código real Intel IA-32; (iii) oferece vários modos de depuração passo-a-passo distintos em cada fase da compilação; e (iv) está inserido em um ambiente integrado com sistema operacional, processador e dispositivos de E/S, permitindo uma maior amplitude de investigação e aprendizado.

3. Processo de compilação

Um compilador é dividido internamente em fases, que definem as etapas funcionais do processo de compilação. Embora na prática algumas dessas fases sejam executadas de maneira intercalada, elas podem ser efetivamente codificadas separadamente.

Com efeito, um compilador utiliza uma arquitetura de software denominada *pipe*. Um programa nessa arquitetura consiste em uma seqüência de componentes c_0, c_1, ..., c_n tal que a entrada para um componente c_x, $x>0$ é a saída do componente $x - 1$ e a entrada para o componente c_0 são os dados a serem processados [13]. No caso específico do compilador, os componentes correspondem às fases do processo de compilação.

O analisador léxico é responsável por ler caracteres do código-fonte e categorizar suas seqüências em unidades lógicas denominadas *tokens* para serem utilizadas por outras partes do compilador, como o analisador sintático. De maneira simplificada, a análise léxica se assemelha à atividade de soletrar. Existem três categorias básicas de *tokens*.

A primeira é a de palavras reservadas, como WHILE ou DO que representam as cadeias “while” e “do” na linguagem C. A segunda categoria é a de símbolos especiais, como MAIOR ou MENOR que representam os caracteres “>” e “<” respectivamente. A terceira categoria de *tokens* representa cadeias múltiplas de caracteres. Exemplos dessa categoria são ID e NUM, que usualmente representam um identificador e um número, seguindo as regras lexicais definidas no projeto da linguagem. Uma peculiaridade da terceira categoria é que uma quantidade potencialmente infinita de caracteres pode ser classificada como um *token*. Para diferenciá-los, é necessário manter a cadeia de caracteres específica, denominada lexema, atrelada ao seu respectivo *token*.

Embora a tarefa do analisador léxico seja converter o código-fonte em uma seqüência de *tokens*, ele raramente faz isso de uma só vez [1, 9]. Como alternativa, o analisador léxico fica subordinado ao analisador sintático, que é responsável por requisitar o próximo *token* quando necessário.

A tarefa do analisador sintático é determinar a estrutura sintática de um programa a partir dos *tokens* passados pelo analisador léxico e gerar uma árvore de análise sintática, também chamada de árvore sintática, que represente essa estrutura. Durante a análise sintática, o analisador léxico é invocado sob demanda do analisador sintático à medida que a análise progride.

A estrutura da árvore sintática depende intrinsecamente da estrutura específica da linguagem sendo compilada. Essa árvore é definida como uma estrutura de dados dinâmica, em que cada nó é composto por um registro contendo vários campos que podem ser utilizados durante todo o processo de compilação. A árvore sintática pode ser considerada o principal artefato do processo de compilação, pois tal estrutura é utilizada e incrementada nas fases de compilação subseqüentes.

A análise semântica é responsável por garantir que um programa está coerente com as regras semânticas da linguagem de programação específica, verificando sua correção e viabilidade para execução. Em uma linguagem com tipos estáticos, como a linguagem C, a análise semântica também é responsável pela construção e manutenção da tabela de símbolos, utilizada para acompanhar a definição dos nomes em um programa, a fim de possibilitar a inferência e verificação de tipos em expressões e declarações. As verificações mais comuns da análise semântica são: verificação de compatibilidade de parâmetros formais e atuais (tipos e quantidade), compatibilidade de tipos em instruções de atribuição, determinação de expressões constantes, verificação de declaração antes do uso e regras de escopo.

O gerador de código intermediário é responsável por gerar uma representação intermediária ao código-final a partir da árvore sintática e da tabela de símbolos, tal que essa representação se aproxime do código-alvo. A utilização de código intermediário é especialmente útil quando o objetivo do compilador é produzir código eficiente, que requer uma grande quantidade de análise das propriedades do código-alvo [9]. O código intermediário também pode ser utilizado para se obter um compilador que possa gerar código-alvo compatível com diversas arquiteturas. Isso pode ser obtido se o modelo de código intermediário for independente de arquitetura.

O gerador de código-alvo é responsável por traduzir o código intermediário em código-alvo. Para realizar essa tarefa deve-se conhecer a localização de todas as variáveis e temporários, além do código para a manutenção do ambiente de execução. Uma questão importante nessa fase é a alocação de registradores e a manutenção das informações sobre o uso dos mesmos. Se o código intermediário utilizado for independente de arquitetura, o gerador de código-alvo é o único componente do compilador que precisa ser reescrito para prover portabilidade.

4. Visão geral dos módulos do SOIS

Em sua versão atual, o SOIS possui módulos para a simulação de processador e sistema de E/S, sistema operacional, montador e compilador.

O módulo Processador simula uma arquitetura superescalar completa. Esse modelo de processador implementa um *pipeline* de 6 estágios e realiza execução fora-de-ordem e especulativa, com previsão de desvios e escalonamento dinâmico de instruções baseado no algoritmo de Tomasulo [14]. A arquitetura alvo do processador simulado é baseada na IA-32 desenvolvida pela Intel, a qual tem sido usada na maioria dos microprocessadores existentes hoje no mercado, tais como o Pentium da Intel e Athlon da AMD. O conjunto de instruções suportado é um subconjunto do padrão da IA-32, contendo as principais instruções de lógica, aritmética, desvio, memória e E/S. O simulador interage com um sistema de E/S também simulado, incluindo teclado, disco rígido, barramento, interrupção e *clock*. O simulador é totalmente parametrizado, configurável e possui contadores de acessos, o que permite simular diferentes cenários de execução e avaliar o desempenho.

Por meio de uma interface gráfica com o usuário é possível visualizar o comportamento funcional e o estado dos componentes durante a execução dos programas, assim como a obtenção de dados estatísticos e quantitativos importantes para a avaliação de desempenho sobre diferentes configurações.

Sua interface gráfica é amigável e possui recursos além daqueles encontrados em simuladores comumente usados, tais como visualização das dependências entre as instruções, o estado de execução das unidades funcionais e o conteúdo da memória *cache*, constituindo-se em uma poderosa ferramenta de ensino, aprendizagem e avaliação de desempenho de arquiteturas superescalares. [5]

O módulo Sistema Operacional do SOIS [5] tem como principal objetivo demonstrar a interação existente entre o hardware e software básico presente no módulo Processador. Para facilitar a compreensão do ambiente, o sistema operacional é estruturado de forma que seja possível a depuração em diversos níveis de detalhamento de seus principais componentes funcionais, tais como escalonador de processos, tratador de interrupções, gerenciador de memória e de

teclado, para que assim todos os módulos do SOIS se relacionem da forma mais real possível.

Procurando manter a fidelidade da simulação do SOIS e um ambiente de fácil compreensão, o sistema operacional simulado implementa alguma de suas rotinas básicas utilizando os métodos das classes do módulo do sistema operacional. Para melhorar o desempenho, algumas das rotinas básicas foram escritas em código Assembly do processador simulado. Esses dois modos distintos implementam um ambiente real, fazendo com que seja possível que o usuário desenvolva as rotinas do sistema operacional em linguagem de montagem e analise como é o comportamento do sistema operacional simulado.

O SASM é o montador do SOIS. Desenvolvido inicialmente como um passo precursor para o desenvolvimento do SCC, o SASM contempla todas as instruções suportadas pelo módulo Processador do SOIS. Assim como o SCC, o SASM também foi concebido com o objetivo de auxiliar no ensino [8], principalmente na disciplina de arquitetura de computadores. Seus modos de depuração permitem que o aluno visualize passo-a-passo todo o processo de montagem e, principalmente, evidenciam as peculiaridades de uma arquitetura CISC. De forma semelhante, foram implementadas facilidades voltadas ao ensino no SCC, que serão descritas na Seção 6.

5. Compilador SCC

O SCC é um compilador para a linguagem C proposto para o ambiente SOIS. Foi desenvolvido em C sob a plataforma GNU/Linux, utilizando a abordagem estruturada, implementando as diferentes fases de maneira independente, prezando a utilização de subprogramas simples e fáceis de modificar.

O processo de compilação escolhido é o proposto por Louden [9] e revisado na Seção 3. Basicamente, a compilação é feita em cinco fases: análise léxica, análise sintática, análise semântica, geração de código intermediário e geração de código final. A etapa de otimização de código, que é intermediária entre a geração de código intermediário e a geração de código final, será implementada em um trabalho posterior.

A análise léxica do SCC foi implementada com o auxílio do gerador de analisadores léxicos *flex* [7]. O *flex* é um programa que recebe como entrada um arquivo com um conjunto de expressões regulares. Para cada expressão regular, pode-se associar um *token* específico. Também é possível recuperar o lexema de cada *token* reconhecido. No caso do SCC, as convenções léxicas da linguagem C foram mantidas, exceto pela remoção de algumas palavras-chave referentes a funcionalidades que não foram contempladas pela proposta do ambiente SOIS. Essas funcionalidades serão comentadas adiante nesta seção.

A análise sintática foi implementada com o auxílio do gerador de analisadores sintáticos *bison* [3], que faz parte do conjunto de ferramentas GNU. O *bison* recebe como entrada um arquivo contendo a gramática da linguagem, juntamente com um conjunto de ações específicas, denominadas ações embutidas, a serem executadas em certos pontos da análise sintática.

O *bison* gera analisadores sintáticos ascendentes LALR(1). Uma peculiaridade dos analisadores ascendentes é existência de uma pilha de valores que é mantida conforme a análise sintática progride. Cada símbolo gramatical do lado direito das produções possui uma variável nessa pilha, que é desempilhada toda vez que ocorre uma redução. No lugar desses símbolos é empilhado o símbolo do lado esquerdo da produção. A atribuição de um valor a cada símbolo gramatical deve ser feita manualmente, por meio da utilização de ações embutidas.

No SCC, uma árvore sintática é montada utilizando esse mecanismo da pilha de valores. Para isso, o tipo da pilha de valores foi definido como nós de árvore sintática. Então, para cada regra gramatical foram criadas ações embutidas utilizadas para alocar e ligar os nós da árvore sintática.

A gramática utilizada pelo SCC é uma versão modificada do padrão ANSI C. As modificações foram necessárias devido a limitações no conjunto de instruções do módulo Processador do SOIS, que não simula todas as instruções necessárias para a implementação da linguagem ANSI C por completo. As principais modificações foram: somente os tipos primitivos *int* e *void* são válidos (*structs* ainda são permitidos); declarações de função só utilizam o padrão "*tipo nome (parâmetros) {...}*". Mesmo com tais modificações, é possível implementar todos os programas que o SOIS pode executar.

Foi implementado um procedimento denominado *parse*, que chama o analisador sintático, extrai a árvore sintática gerada durante a análise sintática e a retorna ao programa principal do compilador. Mediante a opção –a na execução do compilador, a árvore sintática é apresentada, mostrando, de forma didática, a estrutura sintática do programa sendo compilado.

A análise semântica foi dividida em dois grupos: (i) a geração, manutenção e atualização da tabela de símbolos; e (ii) outras verificações. Para cada grupo de verificações semânticas, uma passada pela árvore sintática é necessária. Embora seja possível fazê-la em uma única passada, foi decidido separá-las para simplificar o processo de análise semântica, facilitando

a compreensão do processo pelo usuário do compilador, principalmente durante a depuração.

A tabela de símbolos é construída em um percurso pré-ordem da árvore sintática. Com efeito, a tabela de símbolos gerada pelo analisador semântico do SCC é na verdade uma "árvore de símbolos". Ela recebe esse nome, pois, para cada escopo, uma nova tabela de símbolos é criada e ligada em seu pai estático. Ao reconhecer um novo símbolo, o mesmo é inserido em seu devido escopo. Para verificar se um símbolo está definido em certo escopo do programa, basta fazer uma busca seqüencial, seguindo o apontador para o nó estático de cada tabela de símbolos. Ao contrário da abordagem proposta em [9], que propõe um sistema de pilhas, a árvore de símbolos fica preservada para uma análise pós-compilação, o que é vantajoso no uso do compilador para fins educacionais. Mediante a opção -t, o compilador mostra a tabela de símbolos detalhando o aninhamento de escopos e os atributos de cada símbolo.

As demais verificações semânticas são realizadas em um único percurso pós-ordem pela árvore sintática. As verificações implementadas são: verificação de compatibilidade de parâmetros formais e atuais (tipos e quantidade), compatibilidade de tipos em instruções de atribuição, determinação de expressões constantes, verificação de declaração antes do uso e regras de escopo.

Na geração de código intermediário, a representação intermediária utilizada foi o código de três endereços [1, 9]. Sua implementação consiste em um vetor dinâmico de quádruplas, onde cada elemento é uma instrução de código intermediário. As quádruplas são compostas de um campo que indica a operação a ser realizada naquela instrução e os demais campos indicam os operandos referenciados. Essa representação é independente de arquitetura e, portanto, é possível que sejam feitas as otimizações independentes de arquitetura. A opção -i faz com que o compilador mostre o código intermediário gerado.

A geração de código-alvo foi implementada segundo a proposta de [1]. Esse algoritmo trabalha sobre a representação de três endereços e mantém atualizados os descritores de registradores e de endereços, para a localização dos nomes e temporários.

Como nem todas as instruções e modos de endereçamento estão disponíveis no processador simulado do SOIS, algumas instruções e soluções propostas originalmente tiveram que ser substituídas por seqüência de instruções equivalentes.

A interface escolhida para o SCC foi uma interface de linha de comando simples, com um sistema de ajuda intuitivo e de fácil compreensão, que pode ser invocado pela opção –?.

O modo de depuração permite que o usuário acompanhe, passo-a-passo, toda a execução do compilador e as interações entre seus componentes. Isso dá ao usuário, primordialmente acadêmicos de graduação cursando disciplinas de Compiladores, Linguagens de Programação e Arquitetura de Computadores, uma ótima oportunidade para verificar na prática como um compilador funciona. A descrição dos modos de depuração aparece na próxima seção.

6. Experimentos e Resultados

Com a geração do código-alvo, o SCC atingiu um de seus objetivos, o de gerar código que pode ser montado pelo SASM, montador do SOIS, permitindo que programas em C possam ser executados pelo simulador do ambiente.

Seu segundo objetivo, o de atuar como ferramenta de ensino, é alcançado por meio de funcionalidades extras apresentadas nesta seção. Tais funcionalidades devem auxiliar no entendimento de áreas específicas da construção e execução de um compilador real.

A Figura 1 apresenta um trecho de código C que será utilizado como base para os exemplos discutidos ao longo desta seção. Esse trecho de código mostra duas funções: *pot* e *main*. A função *pot* recebe dois parâmetros: base e exp, que são utilizados para calcular a potência $base^{exp}$ recursivamente. A função *main* simplesmente invoca a função *pot* para calcular 5^a onde a = 2.

A capacidade de visualizar os artefatos intermediários do processo de compilação é uma das funcionalidades implementadas no SCC. Para cada uma das fases de compilação, pode-se requisitar, via linha de comando, que seja mostrado o artefato correspondente. Todos os artefatos solicitados são apresentados ao final da compilação.

```
int pot (int base, int exp){
   if(!exp)
      return 1;
   return base * pot(base, exp-1);
}

int main (int argc, void **argv){
   int a, b;
   a = 2;

   b = pot(5, a);

   return b;
}
```

Figura 1. Código-exemplo

O artefato principal da fase de análise sintática é a árvore sintática abstrata. Nela é mostrada a estrutura sintática do programa, permitindo que o aluno

visualize o mapeamento de um programa-fonte em sua árvore sintática correspondente. A Figura 2 mostra um trecho de uma árvore sintática gerada pelo SCC a partir do código mostrado na Figura 1.

```
Análise sintática finalizada. ASA: 0x805c8c0
(0  ) TIPO_NÓ: T_FUNC
   (0  ) TIPO_NÓ: T_T_TIPO TIPO: INT (291)
   (1  ) TIPO_NÓ: T_FUNC_DECL
      (0  ) TIPO_NÓ: T_ID ID: pot
      (1  ) TIPO_NÓ: T_PARAMETROS
         (0  ) TIPO_NÓ: T_PARAMETRO
            (0  ) TIPO_NÓ: T_T_TIPO TIPO: INT (291)
            (1  ) TIPO_NÓ: T_ID ID: base
         (-->) TIPO_NÓ: T_PARAMETRO
            (0  ) TIPO_NÓ: T_T_TIPO TIPO: INT (291)
            (1  ) TIPO_NÓ: T_ID ID: exp
   (3  ) TIPO_NÓ: T_COMPOUND_ST
      (1  ) TIPO_NÓ: T_IF
         (0  ) TIPO_NÓ: T_UN_OP UN_OP: !
            (0  ) TIPO_NÓ: T_ID ID: pot
         (1  ) TIPO_NÓ: T_RETURN
            (0  ) TIPO_NÓ: T_CONST_N CONST_N: 1
      (-->) TIPO_NÓ: T_RETURN
         (0  ) TIPO_NÓ: T_BIN_OP OP: *
            (0  ) TIPO_NÓ: T_ID ID: base
            (1  ) TIPO_NÓ: T_ATV
               (0  ) TIPO_NÓ: T_ID ID: pot
               (1  ) TIPO_NÓ: T_ID ID: base
               (-->) TIPO_NÓ: T_BIN_OP OP: -
                  (0  ) TIPO_NÓ: T_ID ID: exp
                  (1  ) TIPO_NÓ: T_CONST_N CONST_N: 1
```

Figura 2. Árvore sintática

Como se pode notar, a árvore sintática manteve a estrutura sintática do código intacta. Com a escolha correta dos nós, criou-se uma relação de hierarquia entre os elementos sintáticos, permitindo uma melhor visualização da estrutura sintática do código.

A relação hierárquica entre os nós é dada pela identação, ou seja, um nó com uma identação maior fica mais abaixo na árvore sintática. A relação entre um nó pai e um nó filho pode ser verificada pelo número no início de cada linha. Por exemplo, o número 1 indica que aquele nó é o filho número 1 do nó com um nível de identação inferior a ele. No exemplo, o nó T_PARAMETRO com o número 0 é filho de T_PARAMETROS, que por sua vez é filho de T_FUNC_DECL, que por sua vez é filho de T_FUNC. A relação de irmandade entre os nós é mostrada por uma → ao invés de um número. Esse símbolo indica que esse nó é irmão do nó anterior de mesma identação.

Outro artefato interessante que pode ser bem explorado para o entendimento completo de um compilador é a tabela de símbolos, que neste trabalho é chamada de árvore de símbolos devido à sua estruturação. A Figura 3 mostra a árvore de símbolos relativa ao programa da Figura 1.

```
-----ARVORE DE SIMBOLOS-----


---------------------------

Escopo = global
        Símbolo: main
                Tipo: INT
                Flags: FUNÇÃO
        Símbolo: pot
                Tipo: INT
                Flags: FUNÇÃO

---------------------------

        Escopo = pot
                Símbolo: exp
                        Tipo: INT
                        Flags: PARÂMETRO
                Símbolo: base
                        Tipo: INT
                        Flags: PARÂMETRO

---------------------------

        Escopo = main
                Símbolo: a
                        Tipo: INT
                        Flags: VARIÁVEL
                Símbolo: b
                        Tipo: INT
                        Flags: VARIÁVEL
                Símbolo: argc
                        Tipo: INT
                        Flags: PARÂMETRO
                Símbolo: argv
                        Tipo: VOID
                        Flags: PARÂMETRO PONTEIRO(Prof = 2)
```

Figura 3. Árvore de símbolos

Conforme explicado na Seção 5, a árvore de símbolos é composta de nós que são tabelas de símbolos. Cada nó nessa árvore representa um escopo e as identações representam as relações entre os escopos. No exemplo, o escopo global (que é sempre a raiz da árvore de símbolos) possui somente dois símbolos: *main* e *pot*, que são os dois símbolos que representam as funções definidas nesse escopo. No escopo *pot*, verifica-se que os parâmetros exp e base estão corretamente inseridos na tabela. Pela identação, nota-se que *pot* é um escopo aninhado em global. Já no caso do escopo *main*, nota-se que está no mesmo nível de *pot*, ambos aninhados em global.

Para verificar os nomes visíveis em um determinado escopo na representação utilizada na Figura 3, basta consultar a tabela do escopo e suas tabelas ancestrais (no caso do exemplo, a tabela global). Dessa maneira, é possível verificar de maneira simples se um nome está definido em um determinado escopo ou não. Vale ressaltar que cada nó dessa árvore corresponde ao registro de ativação de cada um dos escopos presentes.

De forma similar à estrutura sintática, a estrutura do código intermediário também pode ser visualizada. Nela, cada linha representa uma instrução do código de

três endereços correspondente ao código sendo compilado.

Além da visualização dos artefatos intermediários, o SCC conta com quatro modos de depuração, correspondentes a cada um dos quatro componentes principais do compilador: o analisador sintático (SIN_DEBUG), o analisador semântico (SEM_DEBUG), o gerador de código intermediário (CI_DEBUG) e o gerador de código-alvo (CA_DEBUG). Cada modo de depuração pode ser acionado de forma independente na linha de comando, podendo ser executados concorrentemente.

No modo de depuração do analisador sintático (SIN_DEBUG), as ações do analisador sintático ascendente LALR (1) são mostradas passo-a-passo, permitindo que o aluno acompanhe todo o processo de análise sintática, além da construção da árvore sintática abstrata.

Por apresentar grande quantidade de detalhes, esse modo de depuração foi dividido em níveis 1 e 2, que indicam o nível de detalhamento a ser mostrado. No nível 1, somente as reduções do analisador sintático são mostradas, focando a estrutura gramatical da linguagem e na construção da árvore sintática. No nível 2 são mostradas todas as mensagens do nível 1, além das mensagens relativas ao processo de funcionamento do analisador sintático LALR (1). Esse modo de depuração pode ser acionado com a opção –dsinx onde x determina o nível de detalhe desejado.

O modo de depuração do analisador semântico (SEM_DEBUG) é responsável por mostrar o funcionamento do analisador semântico do SCC e é dividido em duas partes. A primeira parte, relativa à construção da tabela de símbolos é denominada nível 1. A segunda parte, denominada nível 2, contém todas as mensagens do nível 1, além de mostrar a travessia feita na árvore sintática para a realização das verificações semânticas. Durante a travessia, anotações na arvore sintática são feitas, terminando o processo com uma nova estrutura de árvore sintática, denominada árvore sintática abstrata anotada. Nessa estrutura, informações semânticas, como o tipo de dados de um nó, são derivadas e devidamente guardadas nos nós correspondentes da árvore. A Figura 4 mostra um trecho dessa árvore correspondente ao código da Figura 1.

```
(-->) TIPO_NO: T_RETURN (0x805e900)
   (0  ) TIPO_NÓ: T_BIN_OP (0x805e8b8) OP: *
         ATRIBUTOS:
         Tipo: INT
      (0  ) TIPO_NÓ: T_ID (0x805e680) ID: base
            ATRIBUTOS:
            Tipo: INT
      (1  ) TIPO_NÓ: T_ATV (0x805e870)
            ATRIBUTOS:
            Tipo: INT
         (0  ) TIPO_NÓ: T_ID (0x805e6d8) ID: pot
               ATRIBUTOS:
               Tipo: INT
         (1  ) TIPO_NÓ: T_ID (0x805e730) ID: base
               ATRIBUTOS:
               Tipo: INT
         (-->) TIPO_NÓ: T_BIN_OP (0x805e828) OP: -
               ATRIBUTOS:
               Tipo: INT
            (0  ) TIPO_NÓ: T_ID (0x805e788) ID: exp
                  ATRIBUTOS:
                  Tipo: INT
            (1  ) TIPO_NÓ: T_CONST_N (0x805e7e0) CONST_N: 1
                  ATRIBUTOS:
                  Tipo: INT
                  Type Qualifiers: CONST
```

Figura 4. Árvore Sintática Abstrata Anotada

A Figura 5 apresenta parte da verificação de tipos realizada no código da Figura 1. As verificações semânticas descritas na Seção 5 são contempladas nesse modo. O modo de depuração do analisador semântico pode ser acionado com a opção –dsemx onde x define o nível de detalhe desejado.

O modo de depuração do gerador de código intermediário é composto por um nível e mostra o trajeto feito pela árvore sintática para a geração do código intermediário, além das ações executadas em cada nó. Esse modo de depuração pode ser acionado com a opção –dci.

O modo de depuração do gerador de código-alvo também é composto de apenas um nível e mostra como a tradução do código-intermediário para o código-final é feita. Esse modo foca no mapeamento das instruções hipotéticas utilizadas na representação intermediária para as instruções de máquina reais e específicas do simulador do SOIS. Esse modo de depuração pode ser acionado com a opção –dca.

A principal vantagem dos modos de depuração é auxiliar ao aluno na percepção da seqüência de eventos que compõem o processo de compilação. Assim como no SASM [8], o uso dos modos de depuração pelos alunos deve ser, de início, supervisionado pelo professor, para que o máximo de proveito possa ser feito da ferramenta, unindo a vasta teoria de compiladores com seus respectivos algoritmos e estruturas de dados.

```
4 - SEM_DEBUG: (VERIFICAÇÃO DE TIPOS) ID base Declaration Specifiers: INT PONTEIRO? 0
4 - SEM_DEBUG: (VERIFICAÇÃO DE TIPOS) ID pot Declaration Specifiers: INT PONTEIRO? 0
4 - SEM_DEBUG: (VERIFICAÇÃO DE TIPOS) ID base Declaration Specifiers: INT PONTEIRO? 0
4 - SEM_DEBUG: (VERIFICAÇÃO DE TIPOS) ID exp Declaration Specifiers: INT PONTEIRO? 0
4 - SEM_DEBUG: (VERIFICAÇÃO DE TIPOS) CONST Valor: 1 Declaration Specifiers: CONST INT
4 - SEM_DEBUG: (VERIFICAÇÃO DE TIPOS) BIN_OP Declaration Specifiers (T_ID)OP1: INT (T_CONST_N)OP2: INT
--> OP1 Ponteiro? 0(Prof=0) OP2 Ponteiro? 0(Prof=0)
134604576 - SEM_DEBUG: (VERIFICAÇÃO DE TIPOS) ATIVAÇÃO Nome: pot Declaration Specifiers: INT
4 - SEM_DEBUG: (VERIFICAÇÃO DE TIPOS) BIN_OP Declaration Specifiers (T_ID)OP1: INT (T_ATV)OP2: INT
--> OP1 Ponteiro? 0(Prof=0) OP2 Ponteiro? 0(Prof=0)
```

Figura 5. Modo de depuração do analisador semântico (SEM_DEBUG)

7. Conclusão e trabalhos futuros

Usualmente, as disciplinas de Compiladores apresentam métodos de se implementar cada uma das fases citadas. No entanto, como essas disciplinas possuem uma carga teórica alta, os trabalhos propostos normalmente não utilizam todos os recursos aprendidos na teoria. A proposta do SCC é apresentar aos alunos dessas disciplinas um compilador que utilize métodos normalmente empregados no desenvolvimento de compiladores profissionais, especificamente, mostrando desafios que podem aparecer no projeto e na implementação de um compilador para uma arquitetura CISC.

Além dos modos de depuração, o SCC pode ajudar no entendimento do processo de compilação por meio da disponibilização dos artefatos intermediários do processo, como a árvore sintática, a tabela de símbolos, código intermediário e código-final (linguagem de montagem).

Conforme citado anteriormente, a etapa de otimização de código não foi incluída neste trabalho devido ao curto tempo disponível para a execução do mesmo. No entanto, um módulo de otimização pode ser facilmente acoplado ao sistema, já que a arquitetura de software em *pipe* utilizada prevê esse tipo de atualização. A visualização gráfica da árvore sintática e de outros artefatos também pode ser incorporada ao SCC, contribuindo ainda mais para a fixação desses conceitos.

8. Referências Bibliográficas

[1] Aho, A. V.; Sethi, R.; Ullman, J. D.; Compiladores: Princípios, Técnicas e Ferramentas, Rio de Janeiro, Editora Livros Técnicos e Científicos, 1995

[2] Backes, J.; Dahmer, A. C-gen – Ferramenta de Apoio ao Estudo de Compiladores, XIV WEI - Workshop sobre Educação em Computação, Campo Grande, Julho, 2006.

[3] Bison – GNU Parser Generator. Disponível em http://www.gnu.org/software/bison. Acesso em 5/7/2009

[4] Costa, K. A. P.; Silva, L.A.; Brito, T. P., Auxilio no Ensino de Compiladores: Software Simulador como Ferramenta de Apoio na Área de Compiladores, II Simpósio Internacional de Educação – Linguagens Educativas: Perspectivas Interdisciplinares na Atualidade, Bauru, 2008.

[5] Cruz, E. H. M.; Foleiss, J. H.; Assunção, G. P.; Conçalves, R. A. L.. Ferramenta de Simulação de Processador para Ensino de Graduação e Pesquisa Científica. In: Sulcomp, 2008, Criciúma - SC. Sulcomp, 2008.

[6] Eremin, Evgeny A. Educational Model of Computer as a Base for Informatics Learning, International Journal "Information Theories & Applications" Vol.12, Ithea Institute, Bulgágia, 2005.

[7] Flex – The Fast Lexical Analyzer. Disponível em http://flex.sourceforge.net/. Acesso em 10/8/2009

[8] Foleiss, J. H.; Feltrim, V. D.; Gonçalves, R. A. L. SASM: Uma ferramenta para o ensino do processo de montagem e conjunto de instruções CISC In: WEI – XVII Workshop sobre Educação em Computação, XXIX Congresso da SBC 2009, Bento Gonçalves – RS

[9] Louden, K. C.; Compiladores: Princípios e Práticas. São Paulo, Editora Pioneira Thomson Learning, 2004.

[10] Pentium® Processor Architecture Software Developer's Manual, Volumes 1, 2 e 3: Basic Architecture, Instruction Set Reference e System Programming Guide. Intel Corporation. 2004.

[11] Schneider, C. S.; Passerino, L. M.; Oliveira, R. F. Compilador Educativo VERTO: ambiente para aprendizagem de compiladores, CINTED-UFRGS: Novas Tecnologias na Educação, V.3 Nº 2, Novembro, 2005.

[12] Stallman, R. M. *"Using and porting the GNU Compiler Collection"*, Free Software Foundation, ISBN 059510035X

[13] Somerville, I. Engenharia de Software. 8ª edição. São Paulo: Addison Wesley, 2003

[14] Tomasulo, R. M., "Na Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, pp. 25-33, January 1967. (IBM JORNAL, 1967)