INE5426 - Construção de Compiladores

Análise Semântica e Gerador de Código Intermediário

Entrega: 03 de dezembro de 2019 (até 23:55h via Moodle)

Este Exercício-Programa (EP) pode ser realizado por grupos (com até 4 integrantes). Cada grupo deverá executar duas tarefas chamadas de **ASem** (Análise Semântica) e **GCI** (Geração de Código Intermediário). Trabalharemos com a gramática `CCC-2019-2` (que está em LL(1)).

Os resultados obtidos pelos grupos serão avaliados através da apresentação de um **relatório das atividades** desenvolvidas e através da compilação/interpretação/uso/execução da análise semântica (tarefa **ASem**) e do gerador de código intermediário (tarefa **GCI**).

A nota deste EP é $T_2 = \min\{(T_{2.1} + T_{2.2} + \epsilon), 10.0\}$, onde $T_{2.1}$ está definida na seção 5.1, $T_{2.2}$ está definida na seção 5.2 e $\epsilon \in [0, 1]$ depende da participação dos alunos nas aulas de laboratório.

1 Tarefa ASem

A tarefa **ASem** consiste no uso de regras semânticas para:

1. *Construção de uma árvore de sintaxe T* (nós da árvore somente com operadores e operandos);
2. *Inserção do tipo de variáveis na tabela de símbolos*;

Além disso, alguns pontos realizados na análise semântica deverão ser tratados. São eles:

3. A verificação de tipos (em expressões aritméticas);
    - A verificação de tipos deverá acessar o tipo de uma determinada variável em uma tabela de símbolos.
4. A declaração de variáveis por escopo; e
5. A verificação de que um comando *break* está no escopo de um comando de repetição.

Ponto 1. Construção da árvore de sintaxe

A *construção da árvore de sintaxe* deverá ser efetivada através da aplicação de uma SDD L-atribuída. Para isso, faça o seguinte:

- separe as produções da gramática `CCC-2019-2` que derivam *expressões aritméticas* (chame tais gramáticas de EXPA)
- construa uma SDD L-atribuída para EXPA;
- mostre que a SDD anterior é realmente L-atribuída;
- construa uma SDT para a SDD de EXPA;
- construa uma árvore de sintaxe T para expressões derivadas de EXPA;

Obs.: A construção da árvore de sintaxe poderá ser auxiliada por uma ferramenta.

Ponto 2. Inserção do tipo na tabela de símbolos

Na *declaração de variáveis* cada grupo deverá

- separar as produções da gramática `CCC-2019-2` que derivam *declarações de variáveis*;
- construir uma SDD L-atribuída para DEC;
- mostrar que a SDD anterior é realmente L-atribuída;
- construir uma SDT para a SDD de DEC.

Ponto 3. Verificação de tipos

Neste ponto, cada grupo deverá propor uma solução para o problema de *verificação de tipo* que definimos em seguida. Neste problema é dado uma expressão aritmética. Queremos saber se a expressão é *válida*, ou seja, se é possível realizar as operações da expressão considerando os tipos de cada operando. Vamos considerar que uma operação é válida somente se todos os operadores possuem um mesmo tipo. Por exemplo, a expressão `a + b * 4.7` é válida se `a` e `b` possuem o tipo `float`. A expressão `a + b * ‘‘nada’’` é válida se `a` e `b` possuem o tipo `string`.

Dica: É possível utilizar uma árvore de sintaxe para auxiliar na solução deste problema.

Ponto 4. Declaração de variáveis por escopo

Vamos ilustrar este ponto através dos seguintes exemplos.

Suponha que temos um comando de repetição $R2$ aninhado em outro comando de repetição $R1$.

- A declaração de uma variável

  ```
  int a;
  ```

  poderia ocorrer tanto em $R1$ quanto em $R2$.

- A declaração de

  ```
  int a;
  string a;
  ```

  não pode ocorrer em um único escopo.

Dica: Este ponto pode ser tratado usando uma tabela de símbolos por escopo.

Ponto 5. Comandos dentro de escopos

Este ponto refere-se à verificação do comando `break` no escopo de um comando de repetição. Se tal comando não estiver no escopo de um comado de repetição, então um erro semântico deve ocorrer.

2 Tarefa GCI

A tarefa **GCI** consiste na aplicação de regras semânticas para gerar código intermediário para programas escritos na linguagem derivada por `CCC-2019-2`. Cada grupo deverá

- construir uma SDD L-atribuída para `CCC-2019-2`;
- mostrar que a SDD anterior é realmente L-atribuída;
- construir uma SDT para a SDD de `CCC-2019-2`; e
- usar a SDT de `CCC-2019-2` para gerar código intermediário para os comandos;

Obs.: Uma árvore de derivação criada por uma ferramenta poderá ser utilizada para gerar código intermediário.

3 O que deve ser entregue?

A data para entregar o EP é dia 03 de dezembro (até 23:55h via Moodle). Cada grupo deverá entregar um conjunto de arquivos com:

1. um relatório descrevendo as atividades realizadas no trabalho (em PDF);
    - as respostas das tarefas **ASem** e **GCI** devem ser descritas no relatório.
2. um conjunto de arquivos-programas que resolvem as tarefas **ASem** e **GCI**;
3. três programas escritos na linguagem gerada por `CCC-2019-2` (com pelo menos 100 linhas cada, sem erros léxicos, sem erros sintáticos e com chamadas a funções);
4. *Makefile*s (`https://www.gnu.org/software/make/manual/make.html`).
5. um README com informações importantes para a execução apropriada de todos os programas desenvolvidos.

4 Sobre as compilações/interpretações/execuções dos trabalhos

No momento da execução dos programas desenvolvidos por um grupo, a presença de seus integrantes poderá ser necessária para a efetiva avaliação.

5 Sobre o relatório do trabalho

O relatório do trabalho deverá descrever as atividades realizadas pelos membros do grupo.

5.1 O que será avaliado no trabalho?

Chamamos de $T_{2.1}$ a nota para a avaliação do relatório do trabalho. $0 \leq T_{2.1} \leq 5$. Se algum grupo não entregar o relatório, então $T_{2.1} = 0$. A avaliação considerará a organização do texto e a qualidade da descrição das tarefas realizadas.

5.2 O que será avaliado na execução/uso das tarefas ASem e GCI

Chamamos de $T_{2.2}$ a nota para a avaliação da execução/uso das tarefas **ASem** e **GCI**. $0 \leq T_{2.2} \leq 5$. Abaixo listamos itens importantes com relação a essa avaliação.

- A existência de três programas para `CCC-2019-2` com pelo menos 100 linhas cada, sem erros léxicos, sem erros sintáticos e com chamadas a funções (se não existir os três nas condições citadas, então $T_{2.2} = 0$);
- A existência de um README (se não existir, então $T_{2.2} = 0$);
- A existência *Makefile*(s), (se não existir, então $T_{2.2} = 0$);
- A execução correta do(s) *Makefile*(s) (se não executar corretamente, então $T_{2.2} = 0$);
- A compilação/interpretação dos programas (se há erros de compilação/interpretação em algum programa desenvolvido, então $T_{2.2} = 0$);
- A execução dos seus programas em conjunto com ferramentas (se for o caso);

6 Sobre a entrada e a saída dos dados

A entrada dada será um programa escrito na linguagem `CCC-2019-2`. As saídas esperadas (todas para o terminal) são:

- Em caso de sucesso na *compilação*:
  1. uma árvore de sintaxe para cada expressão aritmética do programa dado (a escrita da árvore deverá seguir a varredura `raiz-esquerda-direita` (veja `https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html`);
  2. tabela(s) de símbolos com atributo do tipo para cada identificador;
  3. uma mensagem de sucesso dizendo que as expressões aritméticas são válidas (ponto sobre verificação de tipos);
  4. uma mensagem de sucesso dizendo que a declaração de variáveis por escopo é válida;
  5. uma mensagem de sucesso dizendo que todo `break` está no escopo de um `for` (ponto sobre comandos dentro de escopos);
  6. um código intermediário para a entrada (código de três endereços).
- Em caso de insucesso na *compilação*:
  1. uma mensagem de insucesso, esclarecendo ao usuário o que ocorreu de errado; indique a linha e a coluna no arquivo de entrada onde ocorreu o erro;

Obs.: Em caso de insucesso na *compilação*, pare o processo no primeiro erro que encontrar e desenvolva a mensagem de insucesso baseando-se neste erro.

Observações importantes:

1. Os programas podem ser escritos em C (compatível com compilador gcc versão 7.4.0), C++ (compatível com compilador g++ versão 7.4.0), Java (compativel com compilador javac versão 11.0.4) ou Python (compatível com versão 3.6.8), e deve ser compatível com Linux/Unix.

2. Se for desenvolver em Python, então especifique (no *Makefile* principalmente) qual é a versão que está usando. Prepare seu *Makefile* considerando a versão usada. Por exemplo, se você usou Python 3, então coloque no *Makefile* entradas com `python3`.

3. Exercícios-Programas atrasados **não** serão aceitos.

4. Programas com *warning* na compilação terão diminuição da nota.

5. É importante que seu programa esteja escrito de maneira a destacar a estrutura do programa.

6. O relatório e o programa devem começar com um cabeçalho contendo pelo menos o nome de todos os integrantes do grupo.

7. Coloque comentários em pontos convenientes do programa, e faça uma saída clara.

8. A entrega do Exercício-Programa deverá ser feita no Moodle.

9. O Exercício-Programa é individual por grupo. Não copie o programa de outro grupo, não empreste o seu programa para outro grupo, e tome cuidado para que não copiem seu programa sem a sua permissão. Todos os programas envolvidos em cópias terão nota T_1 igual a ZERO.

Bom trabalho!

Abaixo está a gramática `CC-2019-2` na forma BNF e com produções que derivam chamadas para funções. Ela é fortemente baseada na gramática X++ de Delamaro (veja bibliografia no plano de ensino). Os símbolos terminais de `CC-2019-2` estão na cor amarela. Os terminais não-triviais são somente *ident*, *int_constant*, *float_constant* e *string_constant*. Os símbolos não-terminais de `CC-2019-2` estão em letra de forma. Os demais símbolos (na cor azul) são símbolos da notação BNF. As produções envolvidas com chamadas de função são precedidas por um símbolo $\star$. Consulte o livro de Delamaro para mais informações sobre a notação BNF (seção 2.3 - página 12).

Livro do Delamaro: `http://conteudo.icmc.usp.br/pessoas/delamaro/SlidesCompiladores/CompiladoresFinal.pdf`

$$
\begin{array}{lll}
\star & PROGRAM & \rightarrow (STATEMENT \mid FUNCLIST)? \\
\star & FUNCLIST & \rightarrow FUNCDEF\ FUNCLIST \mid FUNCDEF \\
\star & FUNCDEF & \rightarrow def\ ident(PARAMLIST)\{STATELIST\} \\
\star & PARAMLIST & \rightarrow ((\ int \mid float \mid string\ )\ ident,\ PARAMLIST \mid \\
 & & \quad (\ int \mid float \mid string\ )\ ident)? \\
 & STATEMENT & \rightarrow (VARDECL; \mid \\
 & & \quad ATRIBSTAT; \mid \\
 & & \quad PRINTSTAT; \mid \\
 & & \quad READSTAT; \mid \\
 & & \quad RETURNSTAT; \mid \\
 & & \quad IFSTAT \mid \\
 & & \quad FORSTAT \mid \\
 & & \quad \{STATELIST\} \mid \\
 & & \quad break\ ; \mid \\
 & & \quad ;) \\
 & VARDECL & \rightarrow (\ int \mid float \mid string\ )\ ident\ ([int\_constant])^* \\
\star & ATRIBSTAT & \rightarrow LVALUE = (\ EXPRESSION \mid ALLOCEXPRESSION \mid \\
 & & \quad FUNCCALL) \\
\star & FUNCCALL & \rightarrow ident(PARAMLISTCALL) \\
\star & PARAMLISTCALL & \rightarrow (ident,\ PARAMLISTCALL \mid ident)? \\
 & PRINTSTAT & \rightarrow print\ EXPRESSION \\
 & READSTAT & \rightarrow read\ LVALUE \\
 & RETURNSTAT & \rightarrow return \\
 & IFSTAT & \rightarrow if(\ EXPRESSION\ )\ STATEMENT\ (else\ STATEMENT)? \\
 & FORSTAT & \rightarrow for(ATRIBSTAT;\ EXPRESSION;\ ATRIBSTAT) \\
 & & \quad STATEMENT \\
 & STATELIST & \rightarrow STATEMENT\ (STATELIST)? \\
 & ALLOCEXPRESSION & \rightarrow new\ (int \mid float \mid string)\ ([\ NUMEXPRESSION\ ])^+ \\
 & EXPRESSION & \rightarrow NUMEXPRESSION((\ < \mid > \mid <= \mid >= \mid == \mid\ !=) \\
 & & \quad NUMEXPRESSION)? \\
 & NUMEXPRESSION & \rightarrow TERM\ ((+ \mid -)\ TERM)^* \\
 & TERM & \rightarrow UNARYEXPR((\ * \mid \backslash \mid \%)\ UNARYEXPR)^* \\
 & UNARYEXPR & \rightarrow ((+ \mid -))?\ FACTOR \\
 & FACTOR & \rightarrow (int\_constant \mid float\_constant \mid string\_constant \mid null \mid \\
 & & \quad \mid LVALUE \mid (NUMEXPRESSION)) \\
 & LVALUE & \rightarrow ident(\ [NUMEXPRESSION]\ )^*
\end{array}
$$