INE5426 - Construção de Compiladores

Fase de análise e síntese

Entrega: 27 de junho de 2025 (até 23:55h via Moodle)

Este Exercício-Programa (EP) pode ser realizado por grupos (máx. 5 integrantes). Cada grupo deverá executar as seguintes tarefas:

- Construção de um analisador léxico para uma linguagem (**AL**);
- Construção de um analisador sintático para uma linguagem (**AS**);
- Construção de um analisador semântico para uma linguagem (**ASem**);
- Construção de um gerador de código intermediário (**GCI**); e

Trabalharemos com uma linguagem denominada `ConvCC-2025-1`. Os *tokens* (os terminais da gramática) associados a essa linguagem estão disponível no fim deste texto. Se desejarem, os grupos poderão realizar *pequenas* modificações na linguagem. No entanto, qualquer modificação deverá ser detalhada no cabeçalho dos arquivos que definem o compilador.

A nota deste EP é $T = \min\{10, \sum_{i=1}^{5} T_i\} \times T_6$, sendo T_1 uma nota associada ao analisador léxico (Tarefa **AL** e definida na seção 1), T_2 uma nota associada ao analisador sintático (Tarefa **AS** e definida na seção 2), T_3 está associada ao analisador semântico (Tarefa **ASem** e definida na seção 3), T_4 está associada a geração de código intermediário (Tarefa **GCI** e definida na seção 4), T_5 está definida na seção 6 e T_6 está definida na seção 7.

1 Tarefa AL

A tarefa **AL** consiste na implementação de um analisador léxico para uma linguagem de programação. O analisador léxico precisa necessariamente ler caracter por caracter da entrada e deve ser baseado em diagramas de transição. Uma *tabela de símbolos* deverá ser implementada. Essa tabela deverá ter uma entrada por *token* `Identificador` e deverá manter uma lista de ocorrências do identificador no arquivo (linha e coluna onde ocorre este token). Além disso, essa tabela tem que estar preparada para novas inserções de propriedades de *tokens* do tipo `Identificador`.

Todos os integrantes dos grupos devem dominar qualquer questão relacionada à tarefa **AL**.

O que será avaliado na análise léxica?

Chamamos de T_1 a nota para a avaliação da análise léxica. $0 \leq T_1 \leq 1,5$. Será observada a construção da tabela de símbolos e a execução do analisador léxico lendo caracter por caracter da entrada e baseado em diagramas de transição. Serão inseridos erros léxicos na entrada que deverão ser capturados pelo analisador léxico.

2 Tarefa AS

A tarefa **AS** consiste na implementação de um analisador sintático para uma linguagem de programação. A gramática associada (`CC-2025-1`) **deve** estar em LL(1). O analisador sintático

deve construir uma tabela de reconhecimento sintático uma única vez. Depois disso, o analisador deverá usar essa tabela para demonstrar (ou não) a pertinência da entrada na linguagem gerada pela gramática (`ConvCC-2025-1`). Abaixo há algumas sugestões que podem ajudar a alcançar tal objetivo.

1. `CC-2025-1` está na forma BNF. Coloque-a na forma *convencional* de gramática. Essa gramática na forma convencional será a `ConvCC-2025-1`.
2. Remova recursão à esquerda de `ConvCC-2025-1`, se existir.
3. Fatore `ConvCC-2025-1` à esquerda, se ela não estiver fatorada.
4. Faça `ConvCC-2025-1` ser uma gramática da classe LL(1). É permitido adicionar novos terminais na gramática, caso necessário. Utilize o teorema visto em aula para demostrar que a gramática resultante está em LL(1).
5. Depois que `ConvCC-2025-1` estiver em LL(1), construa a tabela de reconhecimento sintático.

Mais uma vez, todos os integrantes dos grupos devem dominar qualquer questão relacionada à tarefa **AS**.

O que será avaliado na análise sintática?

Chamamos de T_2 a nota para a avaliação da execução da análise sintática. $0 \leq T_2 \leq 2,5$. Será observada a construção da tabela de reconhecimento sintático (uma única vez), e a execução do analisador sintático para `ConvCC-2025-1` em LL(1). Serão inseridos erros sintáticos na entrada que deverão ser capturados pelo analisador sintático. Os grupos poderão utilizar qualquer ferramenta para gerar a tabela de reconhecimento sintático.

3 Tarefa ASem

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

1. *Construção de uma árvore de expressão 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 verificação de identificadores por escopo; e
5. A verificação de que um comando *break* está no escopo de um comando de repetição.

Em seguida detalhamos cada um dos pontos.

Ponto 1. Construção da árvore de expressão

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

- separe as produções da gramática `ConvCC-2025-1` que derivam *expressões aritméticas* (chame tais produções de gramática 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 expressão T para expressões derivadas de EXPA;

Ponto 2. Inserção do tipo na tabela de símbolos

Na *inserção de tipo na tabela de símbolos* cada grupo poderá

- separar as produções da gramática `ConvCC-2025-1` que derivam *declarações de variáveis* (chame tais produções de gramática DEC);
- 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 expressão para auxiliar na solução deste problema.

Ponto 4. Verificação de identificadores 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.

Além disso, nomes de funções e variáveis em um mesmo escopo não podem ser iguais.

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.

Todos os integrantes dos grupos devem dominar qualquer questão relacionada à tarefa **ASem**.

O que será avaliado na análise semântica?

Chamamos de T_3 a nota para a avaliação da execução da análise semântica. $0 \leq T_3 \leq 3,0$. Será observado o desenvolvimento prático de cada ponto destacado na seção 3 (isto é, se a execução do analisador semântico considera os pontos discutidos na seção 3).

4 Tarefa GCI

A tarefa **GCI** consiste na aplicação de regras semânticas para gerar código intermediário para programas escritos na linguagem derivada por `ConvCC-2025-1`. **O código intermediário que deve ser considerado é o código descrito nas aulas teóricas.** Cada grupo deverá

- construir uma SDD L-atribuída para `ConvCC-2025-1` com regras semânticas que gerem código intermediário;
- construir uma SDT para a SDD de `ConvCC-2025-1`; e
- usar a SDT de `ConvCC-2025-1` para gerar código intermediário para os comandos;

Todos os integrantes dos grupos devem dominar qualquer questão relacionada à tarefa **GCI**.

O que será avaliado na geração de código intermediário?

Chamamos de T_4 a nota para a avaliação da execução da geração de código intermediário. $0 \leq T_4 \leq 3,0$. Será observado o desenvolvimento prático da geração de código intermediário (isto é, se a execução do gerador de código intermediário gera de fato um código intermediário como descrito em sala de aula).

5 O que deve ser entregue?

A data para entregar o EP é dia 27 de junho de 2025 (até 08:00h via Moodle). Cada grupo deverá entregar um conjunto de arquivos com:

1. um conjunto de arquivos que definem as fases de análise e síntese de um compilador (pode ser um único arquivo ou vários arquivos);
2. três programas escritos na linguagem `ConvCC-2025-1` (com pelo menos 100 linhas cada, sem erros léxicos, sem erros sintáticos e sem erros semânticos);
3. um *Makefile* para compilação/interpretação/execução do analisador léxico e sintático;
4. um README com informações importantes para a execução apropriada de todos os programas desenvolvidos.

Orientações para construção de um *Makefile*: `https://www.gnu.org/software/make/manual/make.html`

6 Sobre as execuções dos programas desenvolvidos

Primeiro, note que o máximo da soma $\sum_{i=1}^{4} T_i$ é o valor 10. Para os grupos que desenvolveram com sucesso as fases de análise e síntese de um compilador, o tempo de compilação será medido e considerado na nota final. Isto é, se a fase de análise estiver funcionando corretamente e a geração de código intermediário também, então será realizada uma média do tempo de execução da análise léxica, sintática, semântica e geração de código intermediário para entradas com pelo menos 100 linhas e sem erros. O grupo que tiver a média do tempo de compilação mais baixa (tempo médio mais rápido) terá $T_5 = 2,0$. Seja t tal tempo. Para os demais grupos, $T_5 = 2,0 \times \frac{t}{t_g}$, onde t_g é o tempo do outro grupo ($t_g \geq t$).

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

7 Sobre outras questões que serão observadas durante a avaliação

A nota T_6 é uma nota entre 0 e 1 ($0 \leq T_6 \leq 1$). Abaixo listamos situações que fixam valor para T_6.

- A existência de três programas para `ConvCC-2025-1` com pelo menos 100 linhas cada, sem erros léxicos, sem erros sintáticos e sem erros semânticos e com chamadas à funções (se não existir os três nas condições citadas, então $T_6 = 0$);
- A existência de um README (se não existir, então $T_6 = 0$);
- A existência de um *Makefile*, (se não existir, então $T_6 = 0$);
- A execução correta do *Makefile* (se não executar corretamente, então $T_6 = 0$);
- A compilação/interpretação dos programas (se há erros de compilação ou (durante a interpretação de algum programa desenvolvido), então a nota do grupo é $T_6 = 0$);

- Se for identificada alguma cópia entre grupos, então $T_6 = 0$ para todos os envolvidos.
- O valor de T_6 poderá ser uma constante menor que 1 caso algum membro do grupo não contribua com a execução do trabalho.

8 Sobre a entrada e a saída dos dados

A entrada dada será um programa escrito na linguagem `ConvCC-2025-1`. As saídas esperadas (todas para o terminal) são:

- Em caso de sucesso na *compilação*:
  1. uma árvore de expressão 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 as declarações das variáveis por escopo são válidas;
  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 (da mesma forma como foi apresentado nas aulas);
- 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 11.4.0), C++ (compatível com compilador g++ versão 11.4.0), Java (compatível com compilador javac versão 18.0.2), Python 3 (compatível com versão 3.10.12) ou Rust (compatível com rustc versão 1.75.0) e deve ser compatível com Linux/Unix.
2. Se for desenvolver em Python 3, então especifique (no *Makefile* principalmente) qual é a versão que está usando. Prepare seu *Makefile* considerando a versão usada.
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 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 igual a ZERO.

Bom trabalho!

Abaixo encontra-se a gramática `CC-2025-1` na forma BNF. Ela é fortemente baseada na gramática X++ de Delamaro (veja bibliografia no plano de ensino). Os símbolos terminais de `CC-2025-1` estão na cor amarela. Os terminais não-triviais são somente *ident*, *int_constant*, *float_constant* e *string_constant*. Os símbolos não-terminais de `CC-2025-1` estão em letra de forma. Os demais símbolos (na cor azul) são símbolos da notação BNF. 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`

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