UNIVERSIDADE PRESBITERIANA MACKENZIE
- Faculdade de Computação e Informática –

Ciência da Computação
PROJETO --- Compiladores

A proposta desta atividade é a implementação das etapas *front-end* do projeto de uma linguagem de programação, ou seja, as primeiras etapas da construção do compilador de uma linguagem específica.

Você construirá o *front-end* de uma linguagem simplificada aqui chamada de **AlgC** (que se trata de uma adaptação simplificada da linguagem C) e deve utilizar a linguagem C para a implementação do compilador.

Etapas:
(1) Autômato finito
(2) Analisador léxico
(3) Analisador sintático
(4) Analisador semântico

OBS$_1$: todas as etapas devem ser entregues na data determinada.
OBS$_2$: cada entrega deve ser composta por 2 arquivos: código e documentação (ou, simplesmente, o código muito bem comentado).

ETAPA#1 – Gramática Livre de Contexto e Autômatos Finitos

- A partir das definições da linguagem específica e da sugestão de sintaxe (final do arquivo), apresentar as expressões regulares e construir a gramática livre de contexto correspondentes.
- Construir os autômatos para cada classe de elementos do vocabulário terminal da linguagem: letras, dígitos, números, identificadores, operadores, palavras reservadas, pontuação, comentários.
- Testar cada classe gerada.
- Juntar todos os autômatos em um único para que represente todos os elementos.

OBS$_4$:encaminhar o autômato completo e não separadamente para cada padrão definido.

OBS$_5$:utilizar a ferramenta JFLAP **versão 7.0** (www.jflap.org) para a construção dos autômatos.

ETAPA#2 – Analisador léxico

- Desenvolver um programa que realize a análise léxica.
- O analisador léxico deverá ser fiel ao afd resultante da ETAPA#1 e codificado segundo a estatégia baseada no uso de rótulos e comandos `goto`.
- A partir de um arquivo contendo um código (programa) em AlgC, o analisador léxico deverá produzir um arquivo com os ***tokens*** identificados e seus valores correspondentes (se necessário).
- Considerar que todos os ***tokens*** estão separados por um espaço.
- Cada linha do arquivo de saída deve conter um ***tokens*** reconhecido.
- O analisador léxico deve ser capaz de lidar com os erros léxicos encontrados no programa. A mensagem de erro, “ERRO LÉXICO” juntamente com a sequência lexicamente errada, devem ser apresentadas e todo o processo finalizado.

ETAPA#3 – Analisador sintático

- Construir um programa que realize a análise sintática.
- Usar a sequência de tokens retornada pelo analisador Léxico (um de cada vez) em um Analisador Sintático Recursivo Descendente baseado na gramática da linguagem.
- Serão permitidos eventuais ajustes na gramática proposta (para adequá-la para um Analisador Sintático Recursivo Descendente).
- Todos os ajustes feitos na gramática deverão ser publicados em um documento que fará parte da entrega desta ETAPA.
- O analisador sintático deve ser capaz de lidar com erros sintáticos encontrados no programa. A mensagem de erro, "ERRO SINTÁTICO" juntamente com o *token* incorreto, devem ser apresentados e todo o processo finalizado.
- É necessário entregar os analisadores léxico e sintático funcionando em conjunto.

ETAPA#4 – Analisador semântico

- Deve-se construir e/ou atualizar a tabela de símbolos e fazer a checagem de tipos.
- É necessário entregar todos os analisadores desenvolvidos funcionando em conjunto.

CRITÉRIO DE AVALIAÇÃO

- **Etapa#1 – 15%**
- **Etapa#2 – 15%**
- **Etapa#3 – 15%**
- **Etapa#4 – 15%**

OBS$_6$: o desenvolvimento de cada ETAPA deve ser explicado e, se preciso for, exemplificado.

- Documentação interna: comentários claros e objetivos incluídos no código fonte do programa

DATAS IMPORTANTES

- **Etapa#1 – 10/SET/2018**
- **Etapa#2 – 24/SET/2018**
- **Etapa#3 – 22/OUT/2018**
- **Etapa#4 – 19/NOV/2018**

GRUPOS de no máximo 4 alunos. Trabalhos individuais serão aceitos somente se tratarem, adicionalmente, do comando **`switch-case`**, que não é descrito neste documento.

OBSERVAÇÕES COMPLEMENTARES

- O trabalho deverá ser feito na linguagem C (padrão ANSI).
- Os trabalhos entregues por grupos deverão ser publicados no Moodle por um único aluno do grupo.
- Os trabalhos deverão estar documentados com as identificações completas de todos os membros do grupo.
- Os programas entregues deverão executar no DEV C++ para Windows. Para isso, as entregas envolvendo código em C deverão publicar uma pasta "*zipada*" contendo todo o projeto (em particular, deverá conter o arquivo de projeto de DEV C++).
- Não é permitido o uso de nenhuma biblioteca externa que não esteja disponível na instalação básica do DEV C++.
- **Projetos que não executarem no DEV C++ para Windows receberão nota ZERO**.

AlgC

AlgC descreve um subconjunto de uma linguagem estruturada genérica. Por ser um subconjunto, todo programa AlgC pode ser facilmente compreendido, bastando que o aluno tenha conhecimentos básicos de linguagens de programação.

AlgC trata apenas de valores inteiros e booleanos. Os comandos permitidos são aqueles que podem ser usados com os tipos de dados mencionados. Os literais não são considerados para manipulação, apenas nos ***prints*** para mensagem de interação com o usuário.

AlgC é uma linguagem *case sensitive*.

Exemplo de código correto

```
program _correto {
    int  _a, _b, _c ;
    bool _d, _e, _f ;

    void _proc(int _arg) {
        int  _a ;
        _a= 1;
        if (_a<1) {
            _a= 12;
        }
    }

    _a= 2;
    _b= 10;
    _c= 11;
    _a= _b+_c;
    _d= true;
    _e= false;
    _f= true;

    print(_b);

    if (_d) {
       _a= 20;
       _b= 10 * _c;
       _c= _a / _b;
    }
    while (_a>1) {
       if (_b>10) {
           _b= 2;
           _a= _a-1;
       } else {
           _a= _a-1;
       }
    }
}
```

Um programa em AlgC deverá estar codificado em um único arquivo fonte, sem fazer referências a detalhes externos a ele.

ESPECIFICAÇÃO LÉXICA

- **Comentários**
  Aparecem delimitados por "/*" e "*/". Tudo que segue os símbolos"/*" é ignorado pelo MiniAlg até que se encontre "*/".

```
/* comentário */
```

- **Identificadores (variáveis)**
  Podem ser criados somente com letras de **a** até **z** (maiúsculas ou minúsculas) e devem, obrigatoriamente, começar com o caractere **_** ("*underline*").

```
_Value
_companyNumber

_12Val          /* inválido */
Val12           /* inválido */
_Ex$            /* inválido */
company-name    /* inválido */
_company_name   /* inválido */
```

OBS: Todos os identificadores devem ser declarados antes de serem utilizados.

- **Tipos de dados**
  Numerais : Inteiros (tipo **int**)
  Booleano : **false** | **true** (tipo **bool**)

- **Operadores**
  + - / * < > != <= >= == =

- **Delimitadores**

  ( ) , ; { }

- **PalavrasReservadas**

```
program          if          else
void             true        false
int              bool        while
print
```

SINTAXE

1. ⟨programa⟩ ::= **program** ⟨identificador⟩ **{** ⟨bloco⟩ **}**

2. ⟨bloco⟩ ::= **[** ⟨parte declarações de variáveis⟩ **]**
[⟨parte declarações de funções⟩ **]**
⟨comando composto⟩

Declarações

3. ⟨parte declarações de variáveis⟩ ::= ⟨declaração de variáveis⟩ **{** ⟨declaração de variáveis⟩ **}**

4. ⟨declaração de variáveis⟩ ::= (**int** | **bool**) ⟨lista de identificadores⟩ **;**

5. ⟨lista de identificadores⟩ ::= ⟨identificador⟩ [**,** ⟨identificador⟩]

6. ⟨parte declarações de funções⟩ ::= { ⟨declaração de função⟩ **;** }

7. ⟨declaração de função⟩ ::=
void ⟨identificador⟩ ([⟨parâmetros formais⟩]) **{** ⟨bloco⟩ **}**

8. ⟨parâmetros formais⟩ ::= ⟨parâmetro formal⟩ { **,** ⟨parâmetro formal⟩ }

9. ⟨parâmetro formal⟩ ::= (**int** | **bool**) ⟨identificador⟩

Comandos

10. ⟨comando composto⟩ ::= ⟨comando⟩ **;** [⟨comando⟩ **;**]

11. ⟨comando⟩ ::= ⟨atribuição⟩
| ⟨chamada de procedimento⟩
| ⟨comando condicional⟩
| ⟨comando repetitivo⟩
| **print** (⟨identificador⟩) **;**

12. ⟨atribuição⟩ ::= ⟨variável⟩ **=** ⟨expressão⟩ **;**

13. ⟨chamada de procedimento⟩ ::= ⟨identificador⟩ [**(** ⟨lista de parâmetros⟩ **)**]**;**

14. ⟨lista de parâmetros⟩ ::=
[(⟨identificador⟩ | ⟨número⟩ | ⟨bool⟩)
{**,** (⟨identificador⟩ | ⟨número⟩ | ⟨bool⟩) }]

15. ⟨comando condicional⟩ ::=
if **(** ⟨expressão⟩ **)** **{** ⟨comando composto⟩ **}** [**else** **{** ⟨comando composto⟩ **}**]

16. ⟨comando repetitivo⟩ ::= **while** **(** ⟨expressão⟩ **)** **{** ⟨comando composto⟩ **}**

Expressões

17. ⟨expressão⟩ ::= ⟨expressão simples⟩ [⟨relação⟩ ⟨expressão simples⟩]

18. ⟨relação⟩ ::= **==** | **!=** | **<** | **<=** | **>=** | **>**

19. ⟨expressão simples⟩ ::= [**+** | **-**] ⟨termo⟩ { [**+** | **-**] ⟨termo⟩ }

20. ⟨termo⟩ ::= ⟨fator⟩ { (***** | **/**) ⟨fator⟩ }

21. ⟨fator⟩ ::=
⟨variavel⟩
| ⟨número⟩
| ⟨bool⟩
| **(** ⟨expressão simples⟩ **)**

22. ⟨variável⟩ ::= ⟨identificador⟩

Números e Identificadores

23. ⟨bool⟩ ::= **true** | **false**

24. ⟨número⟩ ::= ⟨dígito⟩ { ⟨dígito⟩ }

25. ⟨dígito⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

26. ⟨identificador⟩ ::= _⟨letra⟩ { ⟨letra⟩ }

27. ⟨letra⟩ ::= a | b | ... | z | A | B | ... | Z

EBNF:

{ α **}** = repetição da cadeia a zero ou mais vezes

[α **]** = cadeia α é opcional

(α **|** β **)** = α **|** β **=** α ou β deve ser escolhido

Não terminais aparecem entre ⟨ e ⟩.