Instituto de Ciências Matemáticas e de Computação

Universidade de São Paulo

Antônio Sebastian - 10797781

Bruno Lanzoni Rossi - 4309596

Gabriell Tavares Luna - 10716400

Helbert Moreira Pinto - 10716504

Trabalho 3: Analisador Sintático

Teoria da Computação e Compiladores

Professor Thiago A. S. Pardo

São Carlos, São Paulo

2021

SUMÁRIO

INTRODUÇÃO

A disciplina de Teoria da Computação e Compiladores é extremamente importante por introduzir diversos tipos de linguagens e gramáticas e elucidar o funcionamento de todas as fases de um compilador. Neste segundo trabalho prático, o desafio se dá na implementação de um analisador sintático descendente preditivo recursivo, que é o "carro-chefe" da compilação. Além disso, integrá-lo com o analisador léxico desenvolvido anteriormente.

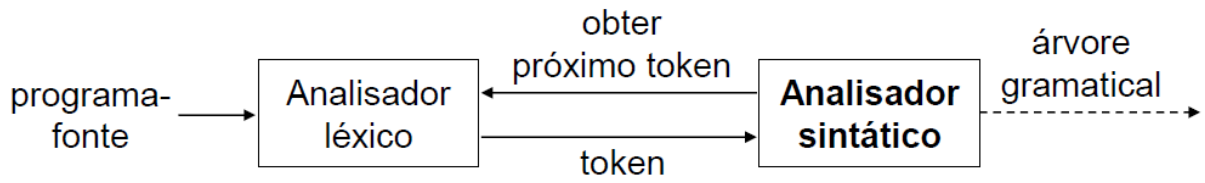
Ao longo das próximas páginas, serão detalhadas informações relevantes para o entendimento do projeto.

REQUISITOS DO PROJETO

O principal objetivo do projeto foi implementar um analisador sintático descendente preditivo recursivo, capaz de ler e interpretar arquivos de texto com o programa-fonte na linguagem P--. O analisador sintático deve realizar o processamento de todo código fonte e executar suas principais funções:

- Verificar a boa formação do programa: verificando quais cadeias pertencem à linguagem
    - Sintaxe, gramática
    - Construção da árvore sintática do programa: implícita ou explícita
- Tratar erros

O analisador léxico implementado anteriormente está integrado e é subordinado ao analisador sintático, que direciona todo o funcionamento do compilador. O sintático verifica se cada cadeia-token solicitado ao léxico está dentro do esperado para atender às regras gramaticais.



Ao final da execução, o programa deve gerar um arquivo de saída indicando eventuais erros léxicos e sintáticos presentes na entrada.

Para a esquematização do projeto, foi necessário mapear a gramática de forma a atender os requisitos de um analisador sintático descendente preditivo recursivo.

Descendente: (Top-down) o analisador parte dos não terminais até os terminais

Preditivo: sabe-se com antemão qual regra aplicar associando os terminais às regras correspondentes

Para isso, é necessário garantir que a gramática siga duas regras: não ter recursão à esquerda e, para cada não terminal, não podem haver regras que começam com o mesmo terminal.

Seguindo esses critérios, pode-se classificar a gramática como LL(1) (Left to right, Leftmost derivation). Isso significa que ela precisará apenas de um símbolo para determinar qual regra aplicar.

Recursivo: apenas na questão da implementação. Não há necessidade de recursão na gramática.

A implementação consiste num conjunto de procedimentos possivelmente recursivos - cada não terminal vai gerar um procedimento.

DECISÕES DE IMPLEMENTAÇÃO

ANALISADOR SINTÁTICO

Para a implementação, o primeiro passo foi garantir que a gramática P-- seguia o formato desejado (LL(1)). No caso de inconsistências, foram realizadas alterações necessárias para que a gramática se adequasse à norma.

Para transformar a gramática P-- em LL(1), foi necessário fazer alterações para o não-terminal <cmd>. O problema consistia em duas regras começando com o mesmo terminal "ident":

<cmd> ::= read (<variaveis>) |
write (<variaveis>) |
while (<condicao>) do <cmd> |
if <condicao> then <cmd> <pfalsa> |
ident := <expressão> |
ident <lista_arg> |
begin <comandos> end

Reescrevendo <cmd> e adicionando <cmd2>, tem-se:

<cmd> ::= read (<variaveis>) |
write (<variaveis>) |
while (<condicao>) do <cmd> |
if <condicao> then <cmd> <pfalsa> |
ident <cmd2> |

begin <comandos> end

<cmd2> ::= <expressão> |
<lista_arg>

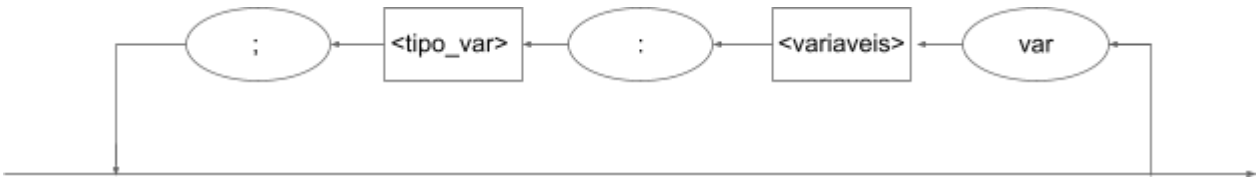
Com a gramática dentro dos requisitos, a linguagem foi mapeada no formato de grafos sintáticos e a partir desses grafos foi feita a tradução para procedimentos.

Durante a implementação, algumas regras foram encapsuladas dentro de outras a fim de minimizar a quantidade de procedimentos do programa.

O mapeamento das regras (LL1) em Procedimentos seguiu os seguintes exemplos::

< program >

< dc_v >



O mapeamento completo pode ser acessado em: https://docs.google.com/presentation/d/1-qwjBGJDnxqy--GRInci2D71X1s00gtW862tXIYYkQM/edit?usp=sharing

TRATAMENTO DE ERROS

Para um tratamento de erro eficaz, nossas funções devem ser capazes de relatar a presença de erros de maneira clara e precisa e ao mesmo tempo devem ser capazes de se recuperar de cada erro para continuar a leitura do programa.

É necessário ser bastante cauteloso e deixar bem definidos quais tipos de erro serão tratados!

Para este projeto, era necessário que nosso tratamento de erros seguisse o **"modo de pânico",** teoricamente o método mais simples e intuitivo de se implementar. Ao encontrar algum erro os passos são:

1) Relata-se o erro.
2) Pulam-se tokens até que um token de sincronização seja encontrado.

Tokens de sincronização são pontos do programa onde é possível retomar a análise sintática com segurança.

A busca por tokens de sincronização deve ser feita sempre que um token esperado não é encontrado.

Para definir os token de sincronização, foram seguidos os seguinte critérios:

- Inicialmente, busca-se pelos seguidores de A (o seguidor local e imediato).

- Caso não encontre, busca-se pelos símbolos seguidores do pai* de A (útil quando tudo dentro do A deu errado).

- Em casos mais radicais (com muitos erros), é interessante ter símbolos extras, pré-definidos, para evitar que seja perdido todo código.

Pode-se destacar como símbolos de sincronização escolhidos a seguinte lista de tokens:

{"simb_program", "simb_begin", "simb_end", "simb_const", "simb_procedure", "simb_read", "simb_write", "simb_while", "simb_if", "simb_else", "simb_do", "simb_to"}

IMPLEMENTAÇÃO DO PROGRAMA

O programa foi implementado em ***C++***, incrementando e aprimorando o código desenvolvido para o primeiro trabalho. Sendo assim, os analisadores léxico e sintático foram programados em classes contidas no programa principal chamado *"compiler.cpp"*. Essas classes foram nomeadas *"lex_an"* e *"sin_an"* e desempenham o papel dos analisadores através de seus métodos e atributos.

Inicialmente o arquivo de texto que deve ser compilado é submetido ao analisador léxico, que realiza o processo de tradução do código, gerando os *"tokens"* e processa os erros léxicos. Os *"tokens"* são

armazenados em uma estrutura de queue (fila) e são passados para o analisador sintático. Os "*tokens*" então são processados pelo analisador sintático com base nos procedimentos sintáticos, que foram implementados de maneira recursiva por meio de métodos da classe. Para tratamento de erros, o analisador sintático utiliza um vetor de símbolos de sincronização e também *"tokens seguidores"*, que foram implementados em uma estrutura de stack (pilha) de strings.

O código foi implementado utilizando de uma abordagem modular dos autómatos, de maneira a facilitar a manutenção e escalabilidade do projeto.

Foi utilizado no desenvolvimento um repositório no GitHub disponível em: https://github.com/helbertmoreirapinto/SCC0605

INSTRUÇÕES PARA COMPILAÇÃO

O programa foi implementado em ***C++*** e nomeado de ***"compiler.cpp"***. Para compilação, foi utilizado o compilador ***gcc*** em diferentes sistemas operacionais, como o Windows e WSL (Windows Subsystem Linux).

O comando de compilação usado foi: ***"g++ compiler.cpp -o out"***, que deve gerar o executável *"out"* na plataforma utilizada.

EXEMPLOS DE EXECUÇÃO

Ao executar o programa, é requerido o nome do arquivo texto que será analisado e deve estar no mesmo diretório do programa. O programa deve imprimir na saída padrão os possíveis erros e relatar o "status da compilação".

Exemplo: Arquivo a ser analisado "meu_programa.txt"

test > meu_programa.txt

```
...
program meu_programa;
var a, 1b: integer;
begin
    read(a,b);
    while (b>a do
        begin
            write(a);
            a=a+1;
        end;
end.
```

Obtendo como saída na execução do programa:

```
gabriell@DESKTOP-K1TOUNS:~/SCC0605/Trab2$ g++ compiler.cpp -o out
gabriell@DESKTOP-K1TOUNS:~/SCC0605/Trab2$ ./out
test/meu_programa
Erro léxico na linha 2: ident mal formado
Erro sintatico na linha: 5, simb_fechar_parentese esperado
Erro sintatica na linha: 8, comparacão não esperada
Quantidade total de erros encontrados: 3
Falha na compilação!
```

Corrigindo os erros contidos arquivo de exemplo, temos o código corrigido:

test > meu_programa_corrigido.txt

```
program meu_programa;
var a, b: integer;
begin
    read(a,b);
    while (b>a) do
        begin
            write(a);
            a:=a+1;
        end;
end.
```

Executando a compilação, obtemos:

```
gabriell@DESKTOP-K1TOUNS:~/SCC0605/Trab2$ ./out
test/meu_programa_corrigido
Quantidade total de erros encontrados: 0
Compilação foi um sucesso!
```

CONCLUSÃO

O desenvolvimento do projeto permitiu visualizar a estrita relação entre a implementação das etapas do compilador, os e autômatos finitos determinísticos e o mapeamento da gramática.

Percebe-se também a eficiência de tal abordagem para escalar o problema, sendo que no caso de se trabalhar com a gramática original do Pascal seria necessário apenas aumentar a possibilidade e a complexidade de comandos e também o dicionário de busca para palavras e símbolos reservadas.

Com a abordagem modular utilizada, o desenvolvimento e implementação do analisador sintático é facilitada de acordo com a estrutura de pilhas e vetores que assim permitem a análise recursiva do sintático, permitindo também uma fácil integração com um analisador semântico, que seria o próximo passo para a construção do compilador.