UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CIÊNCIA DA COMPUTAÇÃO – Campus Chapecó
CCR: Construção de Compiladores

GUSTAVO CAMINEIRO TEREBINTO – 2011100033
LUCAS TRENTINI BORDIN - 2011100024

Construção da compilação de uma linguagem hipotética

Prof. Braulio Mello

FEVEREIRO, 2023

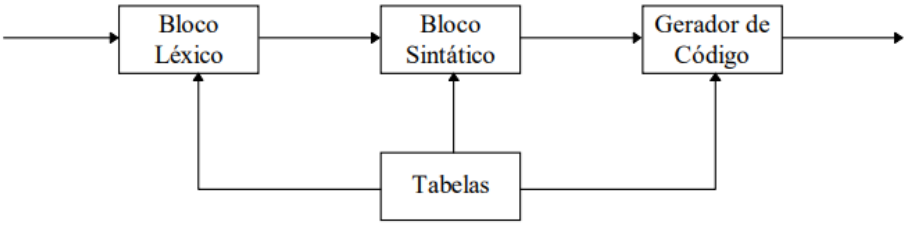
Resumo: Como objetivo, o trabalho busca a construção de um processo de compilação, que ao receber um arquivo com token e outro com o código fonte, faz as etapas para a geração de um código otimizado. Recebendo o arquivo, um programa desenvolvido em linguagem Python lerá o código fonte assim gerando uma fita de saída, neste que estarão contidos os tokens reconhecidos, bem como os erros reconhecidos. No processo seguinte, com uma GLC definida, o software GoldParser nos gera uma tabela de parsing, permitindo sua leitura e consequentemente a criação de um processo de empilhamento, isso baseado na fita de saída. Por fim, a aplicação de algumas modificações na implementação para os processos de semântica e otimização de código, os quais não foram implementados no presente trabalho.

1. INTRODUÇÃO

Este trabalho busca desenvolver um projeto, no âmbito do desenvolvimento e aplicação, consolidando os conhecimentos em Construção de Compiladores. Com o uso de uma linguagem de programação a escolha, o objetivo é projetar um processo de compilação sobre uma linguagem, usando-se dos conceitos teóricos e práticos adquiridos ao decorrer da disciplina. O principal desafio presente neste trabalho objetiva a projeção dos blocos de análise e seu funcionamento, sendo: léxico, sintático, semântico e o gerador de código, isso de modo que siga os princípios teóricos desenvolvidos em sala. Em suma, o desenvolvimento deste busca testar os conhecimentos sobre reconhecimentos de tokens, geração da fita de saída, tabela de parsing e processo de empilhamento, bem como o tratamento de erros durante as etapas.

2. REFERENCIAL TEÓRICO

O processo de um compilador se baseia na leitura de um código fonte, com seus tokens já definidos, de modo a criar uma linguagem de máquina que seja equivalente e que execute de acordo com a lógica do código. Todo esse processo pode ser descrito pelos seguintes blocos: Léxico, Sintático e o Gerador do Código. Esses blocos se interconectam, sempre tendo acesso a uma tabela de símbolos, que serve de referência para o código fonte, onde se encontram informações de variáveis, por exemplo.



2.1. Análise Léxica

Como entrada nessa etapa, temos um código fonte, o qual será composto de uma cadeia de caracteres que seguem a lógica de uma linguagem definida. Nisso, essa cadeia

será fragmentada entre as palavras presentes, de modo a analisar cada uma individualmente. Exemplo: VALOR := A + 100

Para este exemplo, ele identifica de início uma variável *VALOR* junto de uma atribuição *:=*, depois outra variável, *A*, uma atribuição matemática + e uma constante numérica 100. Essa cadeia de caracteres compõe então 5 entidades que se chamam tokens. Em definição, cada token possui um valor atribuído e pertence a um tipo, onde o tipo definirá a função do token.

Para o exemplo anterior, temos: VALOR e A como variáveis, tendo um ponteiro para suas respectivas entradas na tabela de símbolo. Valores como 100, ou outros números, seriam tokens pertencentes ao tipo constante. + e := pertenceriam aos símbolos especiais. Além desses, há algumas palavras reservadas para cada linguagem, como IF e ELSE, por exemplo, que fazem a parte lógica do código.

2.2. Análise Sintática

Também chamado de Parser, é um processo para a geração de derivações de sentenças, baseado em linguagens advindas de uma gramática. O parser então é um mecanismo para criar-se árvores de derivação. Dentro dos analisadores sintáticos, temos dois tipos principais, sendo os *ascendentes*, o enfoque deste trabalho, além dos *descendentes*.

Os ascendentes buscam atingir o primeiro símbolo da gramática, baseando-se na sentença analisada, de modo a decidir a partir da sentença em si ou parte dela, a fim de escolher a produção para a redução. Sendo da família dos analisadores LR, este se baseia em: algoritmo para análise, tabela de parsing, pilha de estados, entrada com sentença e uma GLC. Então no procedimento do analisador LR, compara-se o estado que está no topo da pilha e o símbolo seguinte na entrada, e assim é feita a comparação com a tabela buscando a ação a ser tomada. Sendo elas:

- Parada: conclusão, encerra a análise.
- Erro: erro sintático, procedimento deve contornar o erro ou encerrar.
- Empilhamento: reconhecimento do símbolo, retirando-o da entrada e o empilhando.
- Reduzir: redução de produção, remove da pilha e transforma o token de acordo com o número da redução.

2.3. Análise Semântica

Entre a segunda e terceira fase, para alguns compiladores há um bloco semântico. Processos envolvendo o significado dos símbolos podem ser classificados como parte da semântica, onde a análise pode ver o tipo e até tamanho de um identificador, além da possível adição na tabela de símbolos com as propriedades. Em geral, esse processamento fica responsável por verificar compatibilidade, correspondência de parâmetros, declaração e uso de entidades, entre outras questões de codificação.

2.4. Otimização

Antes da otimização em si, um processo ocorre chamado de código intermediário, que refere-se a converter a árvore de derivação em um código. Esse processo de fragmentação em mais passos permite uma eficiência de código, simplificação do processamento de compilação e multi-interpretação para máquinas. Em contrapartida, esse processamento pode ser mais lento que o código fonte, dado que usa mais passos. Exemplo:

```
A := X * Y * Z
T1 := X * Y
T2 := T1 * Z
A := T2
```

Quanto à própria otimização, se fundamenta principalmente no código intermediário e no código objeto. Para o intermediário, busca-se a eliminação de redundância, sub-expressões comuns, temporários sem uso, entre outros, a fim de atingir um menor tamanho de código, trazendo dinamismo. Para o código objeto, faz-se a conversão das instruções de máquina para outras mais funcionais e rápidas, que consigam melhor uso dos registradores.

3. IMPLEMENTAÇÃO

Como proposto, demos seguimento ao código desenvolvido durante o trabalho da disciplina de LFA, que já gerava o autômato necessário para a interpretação de GRs e conjuntos de tokens. Começamos por definir uma GR e um conjunto de tokens simples, carregados no arquivo “GR.txt”:

```
if
then
else
com
endif
<S> ::= 1<A> | 2<A> | 3<A> | 4<A> | 5<A> | 6<A> | 7<A> | 8<A> | 9<A> | 0<A>
<A> ::= 1<A> | 2<A> | 3<A> | 4<A> | 5<A> | 6<A> | 7<A> | 8<A> | 9<A> | 0<A> | +<B> | -<B> | /<B> | *<B>
<B> ::= 1<B> | 2<B> | 3<B> | 4<B> | 5<B> | 6<B> | 7<B> | 8<B> | 9<B> | 0<B> | =<C>
<C> ::= 1<D> | 2<D> | 3<D> | 4<D> | 5<D> | 6<D> | 7<D> | 8<D> | 9<D> | 0<D>
<D> ::= 1<D> | 2<D> | 3<D> | 4<D> | 5<D> | 6<D> | 7<D> | 8<D> | 9<D> | 0<D> | ε
```

O intuito dos tokens e da GR é construir (na etapa sintática) uma estrutura de condição simples, com a GR gerando expressões no formato [num][op][num]=[num].

Em seguida, implementamos uma função que encontra, no AFD, o conjunto de transições de cada token ou GR e guarda o estado final em um dicionário, para utilização nas próximas etapas:

```
{'if': 'B', 'then': 'F', 'else': 'J', 'com': 'M', 'endif': 'R', '<gram0>': 'W'}
```

3.1 Análise Léxica

Começamos a análise lendo o código carregado no arquivo “codEntrada.txt”. Para cada linha, o código percorre cada um de seus tokens, aplicando as transições de acordo com o AFD e identificando se o token é válido ou não. Sendo válido, o nome do estado que o identifica é impresso na fita de saída, e o token é adicionado na tabela de símbolos juntamente com sua linha, tipo e estado que o reconhece. Caso não seja encontrado, imprimimos na fita um código de erro contendo a linha e o token não identificado. A imagem abaixo contém um exemplo de uma parte do analisador onde pode ocorrer uma adição na fita e na tabela de símbolos ou uma impressão de código de erro na fita:

```python
linhaFim = naoTermAtuais.index(proxEstado)

if tabela[linhaFim][0].startswith("*"):
    fitaSaida += proxEstado + "|"  # adiciona estado na fita de saída
    tipo = ""
    if estado_token[proxEstado] == "exp":
        tipo = "exp"
    else:
        tipo = "word"

    tabSimbolos.append(
        [token, proxEstado, contLinhas, tipo]
    )  # adiciona token (tipo exp ou word) na tabela de símbolos

else:
    # adiciona erro na fita de saída
    fitaSaida += "err:L" + str(contLinhas) + ": " + token + "|"
```

Ao final da análise, é impressa a fita de saída com os estados que identificam cada token, em sequência, e/ou os códigos de erro léxico. Abaixo, dois exemplos de fita, uma com e uma sem erro, respectivamente:

```
Fita de saída: B|W|F|B|W|F|M|err:L4: endfif|J|M|R|$
```

```
Fita de saída: B|W|F|B|W|F|M|R|J|M|R|$
```

Em seguida, também é impressa a tabela de símbolos gerada:

Token	Estado	linha	Tipo
if	B	1	word
1+2=2	W	1	exp
then	F	1	word
if	B	2	word
123/234=32	W	2	exp
then	F	2	word
com	M	3	word
endfif	0	4	err
else	J	5	word
com	M	6	word
endif	R	7	word

3.2 Análise Sintática

Começamos por definir uma GLC para a identificação da estrutura de condição desejada:

```
<C> ::= if <E> then <C> else <C> endif
<C> ::= if <E> then <C> endif
<C> ::= com
<E> ::= exp
```

Nela, o termo “exp” será substituído por uma expressão gerada pela GR no momento de análise.

A partir da gramática, utilizamos o GoldParser para gerar a tabela de parsing LALR em um documento XML para ser interpretado pelo python. Em transcrição livre, a tabela resultante está disposta a seguir:

	if	then	else	endif	com	exp	$	C	E
0	e2				e1			3	
1			r2	r2			r2		
2						e4			5
3							AC		
4		r3							
5		e6							
6	e2				e1			7	
7			e8	e9					
8	e2				e1			10	
9			r1	r1			r1		
10				e11					
11			r0	r0			r0		

Usamos a biblioteca ‘xmltodict’ para converter a tabela XML em um dicionário do python, assim pudemos criar uma função para retornar a ação a ser executada dados o token no início da fita e o estado no topo da pilha.

Por fim, implementamos uma função parser que percorre a fita de saída realizando as ações de empilhamento, redução ou transição. As imagens abaixo mostram um exemplo de código de entrada e do parsing resultante (as imagens de parsing ficam lado a lado mas precisaram ser separadas para caber no documento):

codEntrada.txt

```
1  if 1+234=243 then
2      if 123/234=32 then
3          com
4      endif
5  else
6      com
7  endif
```

```
token: if     | estado: 0  | ação: e2  | fita: ['B', 'W', 'F', 'B', 'W', 'F', 'M', 'R', 'J', 'M', 'R', '$']
token: exp    | estado: 2  | ação: e4  | fita: ['W', 'F', 'B', 'W', 'F', 'M', 'R', 'J', 'M', 'R', '$']
token: then   | estado: 4  | ação: r3  | fita: ['F', 'B', 'W', 'F', 'M', 'R', 'J', 'M', 'R', '$']
token: then   | estado: 5  | ação: e6  | fita: ['F', 'B', 'W', 'F', 'M', 'R', 'J', 'M', 'R', '$']
token: if     | estado: 6  | ação: e2  | fita: ['B', 'W', 'F', 'M', 'R', 'J', 'M', 'R', '$']
token: exp    | estado: 2  | ação: e4  | fita: ['W', 'F', 'M', 'R', 'J', 'M', 'R', '$']
token: then   | estado: 4  | ação: r3  | fita: ['F', 'M', 'R', 'J', 'M', 'R', '$']
token: then   | estado: 5  | ação: e6  | fita: ['F', 'M', 'R', 'J', 'M', 'R', '$']
token: com    | estado: 6  | ação: e1  | fita: ['M', 'R', 'J', 'M', 'R', '$']
token: endif  | estado: 1  | ação: r2  | fita: ['R', 'J', 'M', 'R', '$']
token: endif  | estado: 7  | ação: e9  | fita: ['R', 'J', 'M', 'R', '$']
token: else   | estado: 9  | ação: r1  | fita: ['J', 'M', 'R', '$']
token: else   | estado: 7  | ação: e8  | fita: ['J', 'M', 'R', '$']
token: com    | estado: 8  | ação: e1  | fita: ['M', 'R', '$']
token: endif  | estado: 1  | ação: r2  | fita: ['R', '$']
token: endif  | estado: 10 | ação: e11 | fita: ['R', '$']
token: EOF    | estado: 11 | ação: r0  | fita: ['$']
token: EOF    | estado: 3  | ação: a0  | fita: ['$']
Aceito
```

```
| pilha: [0]
| pilha: [0, 'if', 2]
| pilha: [0, 'if', 2, 'exp', 4]
| pilha: [0, 'if', 2, 'E', 5]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'if', 2]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'if', 2, 'exp', 4]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'if', 2, 'E', 5]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'if', 2, 'E', 5, 'then', 6]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'if', 2, 'E', 5, 'then', 6, 'com', 1]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'if', 2, 'E', 5, 'then', 6, 'C', 7]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'if', 2, 'E', 5, 'then', 6, 'C', 7, 'endif', 9]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'C', 7]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'C', 7, 'else', 8]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'C', 7, 'else', 8, 'com', 1]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'C', 7, 'else', 8, 'C', 10]
| pilha: [0, 'if', 2, 'E', 5, 'then', 6, 'C', 7, 'else', 8, 'C', 10, 'endif', 11]
| pilha: [0, 'C', 3]
```

Caso seja encontrado um código de erro léxico na fita, ou caso seja identificado um erro sintático, a execução do parser para e o erro é informado, juntamente com a linha e o token com problemas:

```
token: exp     | estado: 2  | ação: e4  | fita: ['W', 'err:L2: theen', 'M', 'R', 'J', 'M', 'R', '$']
erro léxico na linha 2: theen
```

```
token: endif   | estado: 8  | ação: None| fita: ['R', 'M', 'R', '$']
erro sintático: 'endif' inesperado na linha 5
```

4. CONCLUSÃO

Neste trabalho, apresentamos a construção e implementação satisfatória de um analisador léxico e um analisador sintático para uma linguagem fictícia determinada. Apesar de não realizarmos os passos de análise semântica, código intermediário e otimização, foi possível implementar e compreender grande parte dos assuntos abordados em aula, tendo coberto as partes essenciais do funcionamento de um compilador de linguagem de programação.

5. REFERENCIAL BIBLIOGRÁFICO

SCHEFFEL, Prof. Roberto M.. Apostila Compiladores I. 2011. Disponível em: https://ramonrdm.files.wordpress.com/2011/04/unisul-apostila-formais-compiladores.pdf. Acesso em: 24 fev. 2023.