TP1 – Parte 2 – Compiladores

UFMG - 2021/2

Nomes: Breno Pimenta ; Lécio Alves
Matrículas: 2017114809 ; 2016065120

1. Introdução:

Este trabalho consiste em implementar um compilador para uma linguagem chamada TIGER, definida a seguir. O trabalho é dividido em três partes e cada uma trata de um componente do compilador. A tabela abaixo mostra os componentes de um compilador, neste trabalho intitulado TP1 – Parte 2 fizemos um **analisador sintático.**

Parte	Função
Analisador Léxico	Analisar o código conforme os símbolos
Analisador Sintático	Verificar se a estrutura do código de acordo com a gramática
Analisador Semântico	Verificar escopos e tipos
Gerador de Código Intermediário	Gerar código intermediário

2. A Linguagem TIGER:

Aqui descrevemos um exemplo de código e a gramática da linguagem TIGER.
Para termos uma ideia de uma entrada válida, abaixo há um exemplo de programa em TIGGER que imprime o somatório de 0 a 10:

```
/* este é um comentário */
let
      val s := 0
      val n := 10
in
      for i := 0 to n do  s := s + i ;
      print(s)
end
```

Temos os seguintes símbolos terminais: *while, for, to, break, let, in, end, function, var, type, array, if, then, else, do, of, nil, além de* ",", ":", ";", "(",

")", "[", "]", "{", "}", ".", "+", "-", "*", "/", "=", "<>", "<", "<=", ">", ">=", "&", "|", ":=" .

Abaixo apresentamos a gramática livre do contexto para a linguagem:

$$
\begin{array}{rcl}
exp & ::= & l\text{-}value \mid \textbf{nil} \mid \texttt{"("}\ expseq\ \texttt{")"} \mid num \mid string \\
 & & \mid \texttt{-}\ exp \\
 & & \mid id\ \texttt{"("}\ args\ \texttt{")"} \\
 & & \mid exp\ \texttt{"+"}\ exp \mid exp\ \texttt{"-"}\ exp \mid exp\ \texttt{"*"}\ exp \mid exp\ \texttt{"/"}\ exp \\
 & & \mid exp\ \texttt{"="}\ exp \mid exp\ \texttt{"<>"}\ exp \\
 & & \mid exp\ \texttt{"<"}\ exp \mid exp\ \texttt{">"}\ exp \mid exp\ \texttt{"<="}\ exp \mid exp\ \texttt{">="}\ exp \\
 & & \mid exp\ \texttt{"\&"}\ exp \mid exp\ \texttt{"|"}\ exp \\
 & & \mid type\text{-}id\ \texttt{"\{"}\ id\ \texttt{"="}\ exp\ idexps\ \texttt{"\}"} \\
 & & \mid type\text{-}id\ \texttt{"["}\ exp\ \texttt{"]"}\ \textbf{of}\ exp \\
 & & \mid l\text{-}value\ \texttt{":="}\ exp \\
 & & \mid \textbf{if}\ exp\ \textbf{then}\ exp\ \textbf{else}\ exp \\
 & & \mid \textbf{if}\ exp\ \textbf{then}\ exp \\
 & & \mid \textbf{while}\ exp\ \textbf{do}\ exp \\
 & & \mid \textbf{for}\ id\ \texttt{":="}\ exp\ \textbf{to}\ exp\ \textbf{do}\ exp \\
 & & \mid \textbf{break} \\
 & & \mid \textbf{let}\ decs\ \textbf{in}\ expseq\ \textbf{end}
\end{array}
$$

$$
\begin{array}{rcl}
decs & ::= & dec\ decs \mid \varepsilon \\
dec & ::= & tydec \mid vardec \mid fundec \\
tydec & ::= & \textbf{type}\ id\ \texttt{"="}\ ty \\
ty & ::= & id \mid \texttt{"\{"}\ id\ \texttt{":"}\ type\text{-}id\ tyfields_1\ \texttt{"\}"} \mid \textbf{array of}\ id \\
tyfields & ::= & id\ \texttt{":"}\ type\text{-}id\ tyfields_1 \mid \varepsilon \\
tyfields_1 & ::= & \texttt{","}\ id\ \texttt{":"}\ type\text{-}id\ tyfields_1 \mid \varepsilon \\
vardec & ::= & \textbf{var}\ id\ \texttt{":="}\ exp \mid \textbf{var}\ id\ \texttt{":"}\ type\text{-}id\ \texttt{":="}\ exp \\
fundec & ::= & \textbf{function}\ id\ \texttt{"("}\ tyfields\ \texttt{")"}\ \texttt{"="}\ exp \\
 & & \mid \textbf{function}\ id\ \texttt{"("}\ tyfields\ \texttt{")"}\ \texttt{":"}\ type\text{-}id\ \texttt{"="}\ exp
\end{array}
$$

$$
\begin{array}{rcl}
l\text{-}value & ::= & id \mid l\text{-}value\ \texttt{"."}\ id \mid l\text{-}value\ \texttt{"["}\ exp\ \texttt{"]"} \\
type\text{-}id & ::= & id
\end{array}
$$

$$
\begin{array}{rcl}
expseq & ::= & exp\ expseq_1 \mid \varepsilon \\
expseq_1 & ::= & \texttt{";"}\ exp\ expseq_1 \mid \varepsilon
\end{array}
$$

$$
\begin{array}{rcl}
args & ::= & exp\ args_1 \mid \varepsilon \\
args_1 & ::= & \texttt{","}\ exp\ args_1 \mid \varepsilon
\end{array}
$$

$$
idexps ::= \texttt{","}\ id\ \texttt{"="}\ exp\ idexps \mid \varepsilon
$$

3. Características do analisador sintático.

O analisador sintático é a parte do compilador que faz a checagem de erros sintáticos: erros na colocação de sentenças do programa. Esse analisador é implementado usando uma gramática livre do contexto e um autômato com pilha, ambos são equivalentes. Primeiramente o scanner recebe o texto do código fonte, elimina comentários e espaços e o transforma em tokens, que nada mais são além de tuplas com a forma (<comando>,<endereço>,<valor>) onde <comando> é um token, <endereço> é a posição do token no código e <valor> é o valor atribuído ao comando, como no caso de constantes, variáveis e nomes de estruturas.

O analisador sintático recebe a lista de tokens e a processa utilizando as regras da gramática livre de contexto especificada, ao final ele deve retornar sucesso ao reconhecer a construção do programa; ou erro, caso o programa esteja escrito errado. Erros detectados serão apenas os relacionados à posição dos tokens, outros erros não serão detectados, como os de tipos incorretos.

O método para essa análise se dá da seguinte forma, como o analisador sintático não consegue analisar entradas e sim tokens, como explicado anteriormente, a cada iteração ele fará uma requisição por um token, assim invocando o analisador léxico. Esse analisador léxico, então, processa o fluxo de entrada e retorna o primeiro token que encontrar. Essa invocação com resposta é contínua e terminá quando o analisador léxico identificar o fim do arquivo de entrada ou quando o léxico ou sintático identificar um erro gramatical.

4. Ferramentas usadas:

Todo trabalho foi feito na linguagem C. Essa entrega é um complemento da entrega do analisador léxico, logo, mantivemos o código desenvolvido com o uso da ferramenta Lex, responsável pelo analisador léxico do nosso trabalho.

Para esta parte do trabalho foi usada a ferramenta yacc, ela tem o papel de automatizar o processo de construir o analisador sintático. O arquivo de configuração do software yacc chama-se **tiger.y** e encontra-se na pasta principal. A estrutura deste arquivo é composto de 3 partes delimitadas pelo símbolo *"**%%**"* , assim como no lex, na primeira seção pode ser incluído código

auxiliar em C e definição dos tokens e opções da ferramenta, na segunda colocamos a gramática, e na terceira pode-se colocar mais código em C.

5. Estrutura do programa:

- Arquivo **tiger.l**: Possui o mapeamento dos tokens para o analisador léxico.
- O Arquivo **tiger.y**: O principal arquivo desta entrega, possui a descrição da gramática livre de contexto e a função **main()** do analisador, esta é responsável pela leitura do arquivo de entrada e realização da chamada da sub-rotina **yyparse()** para o processamento de toda a cadeia de entrada com o automato com pilha dado pela gramática.

As ferramentas Lex e Yacc geram arquivos .c que são a implementação do analisador. Portanto basta compilá-los e executar o binário gerado fornecendo um arquivo de código como entrada.

6. Conflitos encontrados durante o desenvolvimento:

Ao tentar gerar o código do analisador com o yacc, um alerta era gerado: "*tiger.y: warning: 1 reduce/reduce conflict*". Esse conflito ocorre por causa de uma regra na gramática cuja redução é ambígua, por exemplo, "a -> b C | b" sendo C um terminal, logo o analisador não sabe qual regra deve ser reduzida para a, gerando então o conflito. Esse alerta se deu na nossa primeira implementação para duas regras do typeid, então criamos outro não terminal **tid** e resolvemos o conflito. Para descobrir onde o conflito se encontra, usamos a opção **-v** do yacc, ele gerará um arquivo **y.output** que contém toda a descrição do autômato e em qual estado existe o conflito.

Já os casos de precedência e associatividade foram todos resolvidos utilizando diretivas do Yacc, como a indicação de **%left** para associativo à esquerda e **%nonassoc** para não associatividade. A precedência pode ser indicada pela ordem que **%left** ou **%nonassoc** aparece, sendo a menor precedência acima; e a maior, abaixo.

7. Modo de usar:

Para gerar o código, pode ser usado o arquivo makefile incluso
O código do TP pode ser utilizado usando o makefile fornecido, uma pequena ajuda será imprimida na tela ao digitar ***make*** .
Para gerar o analisador léxico e executá-lo com todos os códigos fonte de teste da pasta testes, digite ***make testes*** .

Primeiramente o código fonte é impresso na tela; em seguida, a medida que um arquivo é processado, os tokens são gerados e passados ao analisador sintático que vai verificar se a linguagem está de acordo com a gramática e indicar se aceita ou não.

8. Conclusão:

O nosso trabalho teve uma boa dificuldade, principalmente para aprender a usar a interface das ferramentas, feita via variáveis e funções, mas no caso da teoria foi fácil especificar a gramática, pois as modificações foram mínimas, já que a ferramenta yacc fornece opções para definição de associatividade e precedência.

Bibliografia:

- Aho, Sethi, Ullman, **Compilers: Principles, Techniques, and Tools**, Addison-Wesley, 1986. ISBN 0-201-10088-6
- Nieman, Thomas, **A Compact Guide to Lex & Yacc**, ePaper Press. 2001. Disponível em https://www.classes.cs.uchicago.edu/archive/2003/spring/22600-1/docs/lexyacc.pdf