TP1 – Parte 3 – 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 3** fizemos um **Analisador Semântico e um Gerador de Código Intermediário.**

Parte	Função
Analisador Léxico	Analisar o código conforme os símbolos
Analisador Sintático	Verificar a estrutura do código de acordo com a gramatica
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. Estrutura final do compilador TIGER

A seguir temos os módulos que compõe o compilador TIGER:

- **tiger.l** e **tiger.y** : geram os analisadores léxico, sintático e semântico.
- **absyn** : Contém as estruturas que representam a árvore de derivação, como classes de expressões, declarações, variáveis e comandos, além das funções que as manipulam. Nesse módulo também é feita a conferência de tipos.
- **erro** : Implementa apenas uma função para imprimir uma menstagem de erro e terminar o programa.
- **imprimir** : Imprime a árvore de derivação.
- **tabela** : Implementa a estrutura da tabela de símbolos e algumas funções para manipulação destes.
- **tipos** : Interface que contém a declaração de algumas estruturas de C, originalmente elas estavam no módulo absyn mas para evitar inclusão recursiva dos módulos tabela e absyn, foi criado este tipos para serem incluído por ambos.

3.1 Implementação do analisador semântico.

O analisador semântico é a parte do compilador que faz a checagem de erros nos tipos e no escopo de variáveis: erros onde se usa tipos incompatíveis com o lugar onde eles são avaliados e erros quando se tenta incluir uma variável que já existe no escopo atual. Nesta parte final do trabalho introduzimos a **tabela de símbolos**, necessária para o armazenamento dos símbolos do programa, eventual consulta e resolução de conflitos. A tabela de símbolos foi implementada na **forma LINEAR**, por ser mais fácil criar as rotinas de acesso. A declaração da tabela e suas rotinas estão contidas no arquivo **tabela.h** , junto com uma pequena descrição. Todas as funções estão em conformidade com os algorítimos apresentados em aula.

O yacc provê uma interface que facilita a implementação da checagem de tipos. Em sua primeira seção, a *keyword* **%union{}** pode ser definida. Através dela podemos declarar variáveis de diversos tipos em C, primitivos ou não. Essas variáveis serão usadas em conjunto com a **variável global yyval**, para atribuição do valor real e o seu tipo apropriado, diretamente no lex.

Podemos associar os tipos aos tokens facilmente usando a keyword **%type** <union_member> TOK ou **%token** <union_member> TOK cujo parâmetro <union_member> é opcional.

Em cada regra da gramática, é possível obter o valor resultante de cada elemento do lado direito, usando o símbolo **$**.

Tomemos o seguinte exemplo:

A : B C { print($1.nome); $$=criar_estrutura($1, $2); };

Acima vemos **A** derivando **B** e **C**. **$1** representa o resultado de **B**, já **$2** representa o resultado de **C**. **$$** representa o resultado de **A**, que será acessível em outra regra na qual o **A** esteja do lado direito. Os tipos das variáveis obtidas através de **$n** tem que ser previamente definidos através da **%union{}**

O resultado de uma variável **$*n*** vem de reduções de símbolos do lado direito, e **$$** é a variável onde atribui-se um valor para ser passado a outra regra antes da redução para ela. A função **criar_estrutura** pode ser implementada em outro módulo, podendo conter estruturas e operações semânticas e que geram o código intermediário.

Algumas regras como declarações e atribuições de variáveis inserem símbolos na tabela de símbolos, essa tabela é necessária para guardar o estado de variáveis, pois este é um problema que não pode ser resolvido apenas pela gramática livre de contexto. As variáveis devem ser únicas dentro de um escopo, mas podem ser duplicadas em sub-escopos do escopo atual.

Uma vez montada a árvore de derivação, o próximo passo é gerar o código intermediário, descrevemos isso na próxima seção:

3.2 Gerador de código intermediário

Esta é umas das ultimas etapas das arquitetura convencionais de compiladores. O código intermediário é um código padronizado de três operandos que pode ser facilmente traduzido para código de maquina de qualquer arquitetura, pois ele se assemelha a instruções de maquina. Uma vez gerado este código, podemos otimizá-lo opcionalmente e em seguida usá-lo diretamente em maquinas virtuais, no caso de linguagens interpretadas, ou dar ele para o montador e ligador da maquina de destino, caso queiramos o código *binário* do programa.

A geração do código intermediário pode ser feita através de variáveis dentro das estruturas da AST, as estruturas são recursivas, podemos percorrer elas e obter os valores das variáveis e desmembrá-las em operações separadas para ficarem adequadas ao código de três endereços.

Infelizmente não concluímos essa parte do trabalho, pois não deu tempo.

4. Ferramentas usadas:

Todo o trabalho foi feito na linguagem C. Essa entrega é um complemento das entregas anteriores, logo, mantivemos o código desenvolvido com o uso da ferramenta **Lex**, e **yacc** responsável pelos analisadores léxico e sintático, respectivamente.

Na parte anterior do trabalho, a checagem da sintaxe do programa de entrada foi feito com ajuda da ferramenta **yacc**, onde foi possível definir toda a gramática exatamente da forma como é representada em papel. Então o yacc gera o código em C que representa as ações na tabela de ACTION e GOTO.

5. Conflitos encontrados durante o desenvolvimento:

Encontramos dificuldades em fatorar a gramática e mantê-la livre de conflitos, para melhor organizar as chamadas dos procedimentos do módulo **absyn**. Por isso optamos por deixá-la como está. Adaptamos ou criamos procedimentos para melhor representação da estrutura na linguagem que cada regra representa.

6. Modo de usar:

O código do TP pode ser utilizado usando o Makefile fornecido, uma pequena ajuda será imprimida na tela ao digitar ***make*** , recomendamos utilizar o Linux para a execução deste compilador.
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, é impressa a árvore de sintaxe abstrata.

7. Conclusão:

O nosso trabalho teve uma dificuldade considerável, principalmente na definição das estruturas que representem as partes do programa, pois isso envolvia mexer na gramática, o que gerava conflitos. Mas aproveitamos a oportunidade que esse trabalho nos deu para aprender a usar a interface das ferramentas lex/yacc.

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 Acessado em Jan/2022.
- Escalda, G. e Bigonha, M., **Um compilador para a linguagem TIGER usando a Linguagem Java como Ferramenta de Desenvolvimento**, Relatório Técnico do Laboratório de Linguagens de Programação, LLP11/99.
- **lex and yacc program information**, AIX documentation. Disponível em https://www.ibm.com/docs/en/aix/7.2?topic=concepts-lex-yacc-program-information. Acessado em Jan/2022.