Analisadores Léxicos e Sintáticos

INF1022

PUC
RIO

Trabalho Final

Antenor Barros Leal	2011241
Bernardo Ruiz Fernandes	1711180

Rio de Janeiro

2021

ARQUIVOS ENTREGUES

Junto a este relatório está sendo entregue os seguintes arquivos:

- provol-one.l
- provol-one.y
- Makefile
- demo.sh
- Arquivos de teste

O script demo.sh mostra cada arquivo de teste e o resultado, mostrando que o compilador consegue mostrar a posição de cada erro e uma possível solução.

DECISÕES DO TRABALHO

As seguintes decisões precisaram ser tomadas ao longo do projeto para ajudar no desenvolvimento do código e para solucionar problemas previstos:

- Para solucionar o Conflito Shift-Reduce que foi encontrado no desenvolver do projeto, foi decidido usar comandos como "HEADER", "FIM" e "PROGRAMA" para ajudar na delimitação da definição de variáveis de E/S e do programa. O conflito ocorre caso haja um operador igual logo após a definição das variáveis de saída

```
Program
    ENTRADA a b
    SAIDA c d e
c = a
```

  A operação correta seria o analisador reduzir em “SAIDA c d e”, mas outra opção seria um shift e reduzir em “SAIDA c d e c”, o que estaria gramaticalmente errado e

também desencadearia um outro erro de sintaxe, já que “= a” é inválido. Com as modificações propostas, temos:

```
HEADER
    ENTRADA a b
    SAIDA c d e
FIM
PROGRAMA
    c = a
FIM
```

Dessa forma então, quando iniciamos um ambiente que define as variáveis, precisamos escrever "HEADER" no seu início e "FIM" quando é finalizado. Na mesma lógica, para a parte do programa, usamos a label "PROGRAMA", assim, não geramos mais a ambiguidade.

- Ao longo do desenvolvimento tínhamos implementado para o erro uma indicação de coluna onde ele se encontrava. Mas, logo em seguida, quando já estávamos realizando os últimos testes, entendemos que essa informação não iria necessariamente agregar valor à informação do erro, uma vez que o nosso código consegue identificar o erro e avisar que tipo de informação o precede na linha em questão.
  Assim, como já estamos dando uma informação tão assertiva do erro não vimos necessidade de continuar com a informação da coluna.

DESENVOLVIMENTO

Para iniciar o desenvolvimento do projeto criamos os arquivos .l e o .y de uma forma inicial simples para que seja possível ler um arquivo criado, sem ter preocupações com mensagens de erros inicialmente.

Como estamos desenvolvendo mensagens de erros com uma maior finalidade de ajudar o usuário a indicar a linha é essencial para que o programador identifique mais facilmente seu erro.

Para esse tratamento da linha, nossa primeira abordagem foi usar o *yylineno* que já é nativo do .l. Esse comando só precisa ser inicializado com a seguinte linha de código:

```
%option yylineno
```

Colocamos no arquivo do Yacc um *extern int* para que ela possa ser acessada.

Ao longo do desenvolvimento do projeto nos deparamos com uma complicação referente ao fato de em alguns momentos a indicação da linha do erro não estar de acordo com o local que de fato se encontrava.

Após analisar os possíveis causadores desse erro, entendemos que isso nada mais é que uma característica do lookahead do Yacc, o que pode acabar provocando um avanço do arquivo Lex, consequentemente um incremento da variável que guarda o número da linha, sendo que o .y ainda se encontra analisando a linha anterior

Como solução para esse problema acessamos o fórum que se encontra na bibliografia e encontramos uma solução para a indicação correta da linha. Para isso, temos a variável *first_line* inicializada no .l da seguinte forma:

```
#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;
```

E no nosso código do Yacc acessamos usando o comando *@1.first_line* como foi passado como recomendação no material consultado. O @n funciona de maneira parecida ao $n, mas retorna a posição em vez do conteúdo do token “n”. Dessa forma temos as linhas mapeadas corretamente.

Foi implementada a tabela de símbolos para as variáveis de entrada e as variáveis de saída, ou seja, conseguimos avaliar erros de utilização de variável não declarada. Para tal usamos uma lista de variáveis de entrada e outra para as de saída. Para cada variável colocada, o programa procura nesta tabela, caso não encontre, um erro é mostrado. Separamos em entrada e saída, porque, ao final do código gerado, imprimimos o valor de cada variável de saída. As entradas recebem o *argv[]*, na ordem, ao início do código. Consulte a seção EXEMPLOS.

Para tratar os casos em que temos um erro mas o código continua sendo lido e encontramos outro erro, usamos a variável *erroAnterior* para saber a linha do primeiro para indicar na mensagem de erro, acessando da mesma forma que antes a linha, com o *@1.first_line.*

Seguindo o projeto, realizamos ele montando então cada possibilidade de erro na linguagem para poder abordar de forma personalizada cada erro, onde passamos na mensagem de aviso o que está antecedendo o erro. Pelo exemplo abaixo que se refere a um fragmento do código, vemos que quando temos um erro após o usuário definir uma ação de loop com ENQUANTO, e indicamos na mensagem que o erro está nesse ponto:

```
| ENQUANTO error
    {
        printf("Linha %d:\n", @1.first_line);
        printf("  > O erro está após o ENQUANTO\n");
        printf("  > Uso do ENQUANTO: ENQUANTO a FACA cmds FIM\n");
        printf("  > onde a é uma variável e cmds um ou mais comandos.\n");
        char* result = malloc(1);
        *result = '\0';
        $$ = result;
        erroAnterior = @1.first_line;
    }
```

EXEMPLOS

Exemplos da geração de código C quando o arquivo de entrada está gramaticalmente correto.

ENTRADA

```
HEADER
        ENTRADA entrada1 entrada2 entrada3 entrada4
        SAIDA saida1 saida2 saida3
FIM
PROGRAMA
        ZERA(entrada1)
        ZERA(entrada2)
        INC(entrada2)
        INC(entrada2)
        saida3 = entrada2
        saida1 = entrada3
        saida2 = entrada4
        ENQUANTO saida2 FACA
                INC(saida1)
```

```
            DEC(saida2)
        FIM
        ENQUANTO saida2 FACA
            INC(saida1)
            DEC(saida2)
        FIM
        ENQUANTO saida2 FACA
            INC(saida1)
            DEC(saida2)
        FIM

FIM
```

SAÍDA

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
        int entrada1 = atoi(argv[0]);
        int entrada2 = atoi(argv[1]);
        int entrada3 = atoi(argv[2]);
        int entrada4 = atoi(argv[3]);
        int saida1 = 0, saida2 = 0, saida3 = 0;
        entrada1 = 0;
        entrada2 = 0;
        entrada2++;
        entrada2++;
        saida3 = entrada2;
        saida1 = entrada3;
        saida2 = entrada4;
        while(saida2) {
                saida1++;
                saida2--;
        }
        while(saida2) {
                saida1++;
                saida2--;
        }
        while(saida2) {
                saida1++;
                saida2--;
        }
        printf("%d\n", saida1);
        printf("%d\n", saida2);
        printf("%d\n", saida3);
        return 0;
}
```

ENTRADA

```
HEADER
	ENTRADA a b c
	SAIDA a2 b2 c3
FIM
PROGRAMA
	ENQUANTO c FACA
		INC(a)
		INC(b)
		DEC(c)
	FIM
	ENQUANTO a FACA
		ENQUANTO b FACA
			ENQUANTO c FACA
				INC(a2)
				DEC(b2)
				DEC(c)
			FIM
			DEC(b)
		FIM
		DEC (a)
	FIM
	ZERA(a)
	DEC(c)
FIM
```

SAÍDA

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
	int a = atoi(argv[0]);
	int b = atoi(argv[1]);
	int c = atoi(argv[2]);
	int a2 = 0, b2 = 0, c3 = 0;
	while(c) {
		a++;
		b++;
		c--;
	}
	while(a) {
		while(b) {
```

```
                        while(c) {
                                a2++;
                                b2--;
                                c--;
                        }
                        b--;
                }
                a--;
        }
        a = 0;
        c--;
        printf("%d\n", a2);
        printf("%d\n", b2);
        printf("%d\n", c3);
        return 0;
}
```

Para gerar os exemplos que criamos basta rodar o script *demo.sh* que compila a execução dos arquivos de teste. No final tivemos os seguintes resultados:

- **testeVarInexistente.pvl:** para esse teste que foi realizado é esperado uma resposta de que a emtrada2 na linha 7 não existe;
- **testeErroINC.pvl:** para esse teste que foi realizado é esperado uma resposta de que a a função de inicialização está escrita errada na linha 10 não existe;
- **testeErroZERA.pvl:** para esse teste que foi realizado é esperado uma resposta de que a tem um erro na linha 8 já que não tem os parênteses para o *ZERA( );*
- **testeErroENQUANTO.pvl:** o erro nesse caso é não ter o FACA na linha 10;
- **testeErroDEC.pvl:** o erro desse caso está no DEC da linha 10.

BIBLIOGRAFIA

LEX & YACC TUTORIAL, por Tom Niemann

ON COMPILER ERROR MESSAGES: WHAT THEY SAY AND WHAT THEY MEAN, por V. Javier Traver

Fóruns online acessados para sanar dúvidas e obter inspiração:

- https://stackoverflow.com/questions/22407730/bison-line-number-included-in-the-error-messages
- https://developer.ibm.com/tutorials/l-flexbison/
- https://stackoverflow.com/questions/16936140/how-to-fetch-the-row-and-column-number-of-error
- https://stackoverflow.com/questions/62115979/how-to-implement-better-error-messages-for-flex-bison
- https://stackoverflow.com/questions/32472496/methods-in-yacc-to-get-line-no