Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
SCC0217 - Linguagens de Programação e Compiladores
Prof. Dr. Diego Raphael Amancio

Trabalho Prático 2 - Análise Sintática

Bruno Henrique Rasteiro - 9292910
Carlos André Martins Neves - 8955195
Kairo Bonicenha - 9019790
Marly da Cruz Cláudio - 8936885
Tobias Mesquita Silva da Veiga - 5268356

São Carlos
Junho de 2019

Sumário

1 Compilação e execução do analisador sintático

Para gerar os analisadores sintático e léxico foram utilizados os programas:

- **flex** versão **2.6.4** [1]
- **bison** versão **3.3.2** [2]

O programa foi compilado usando **gcc** versão **8.3.0**.

Para compilar o analisador sintático, navegar para a pasta raiz do projeto, a pasta que possui o arquivo `makefile` e rodar o comando:

```
make
```

Dessa forma, o binário do analisador sintático será gerado, chamado `syntactic-analyser`. Para analisar um programa, o texto do programa pode ser passado diretamente para a stream de input `stdin` do analisador:

```
./syntactic-analyser < lalg-program.txt
```

Ou o nome do arquivo do programa pode ser passado diretamente como argumento:

```
./syntactic-analyser lalg-program.txt
```

Em caso de sucesso, nada é impresso na tela e o programa simplesmente finaliza. Em caso de erro, os erros encontrados são impressos na tela.

1.1 Organização do código

O código do programa está na pasta src, que contém três arquivos: `flex.l`; `flex.text.c`; e `bison.y`.

O arquivo `flex.l` define as opções e as expressões regulares que o flex utiliza para gerar a função de análise léxica `yylex`, utilizada pelo parser gerado pelo bison. O arquivo `flex.test.c` define um programa para compilar apenas o analisador léxico; isso foi feito para debugar o lexer gerado e garantir que ele está funcionando da maneira esperada.

O arquivo `bison.y` define os símbolos terminais, os não terminais e as regras de produção da gramática da linguagem LALG, especificadas no padrão yacc. Esse arquivo define também as funções `yyerror` e `main` do analisador sintático.

1.2 Debug do analisador sintático

Para executar o analisador sintático em modo de *debug*, basta rodar o executável `syntatic-analiser` com a flag `--debug`. Desse modo é possível observar todas as operações que o algoritmo de análise léxica está executando, ou seja, leitura dos *tokens*, empilhamentos/reduções e as mudanças de estado no automato. É valido ressaltar que esse modo de *debug* não foi implementado pelo grupo, mas está disponível no bison.

1.3 Descrição do automato gerado

É possível também imprimir na tela uma descrição do autômato do parser rodando o comando `make analyse-grammar`. O output pode ser redirecionado para um arquivo com o comando `make analyse-grammar > arquivo.txt`.

2 Adaptações em relação ao projeto anterior

A principal adaptação feita no analisador léxico foi a modificação dos comandos que eram executados na identificação de um *token*. No projeto anterior, uma mensagem era exibida no terminal quando um *token* era identificado, agora é retornado um valor inteiro (mapeado em um *enumerate*) que será interpretado pelo yacc como o *token* lido do programa.

Outra mudança realizada foi a remoção da função *keyword*, ela tinha como finalidade indicar quando um identificador reconhecido era uma palavra reservada. Para substituir a função foi utilizado várias expressões regulares (uma para cada palavra reservada) diretamente na segunda seção do lex.

3 Tratamento de erros

Nessa seção explicamos como funciona o analisador sintático do Yacc, conforme especificado em [3], e explicamos nossas decisões ao implementar o tratamento de erro do nosso analisador sintático.

3.1 Como funciona o tratamento de erros do Yacc

O Yacc trata erros utilizando o modo pânico. Para fazer uso dessa funcionalidade, é necessário modificar as produções da gramática, inserindo o símbolo reservado *error* onde acredita-se que um erro pode acontecer e que seja possível se recuperar desse erro.

```
dc_c :
   CONST IDENT '=' numero ';' dc_c | %empty |

   CONST error '=' numero ';' dc_c |
   CONST IDENT error numero ';' dc_c |
   CONST IDENT '=' error ';' dc_c ;
```

Quando em um dado estado da pilha encontra-se um *token* inesperado na cadeia de entrada, então ocorreu um erro sintático. Em seguida, o analisador sintático desempilha até cair em um estado que contenha um item LR(0) da forma $A \rightarrow .\, error\, \alpha$, sendo α uma sequência de *tokens* que pode ser vazia.

A partir desse ponto, o analisador entra em modo pânico e consome a cadeia de *tokens* até sincronizar, ou seja, empilhando o primeiro *token* de α. Se α for vazio então o analisador faz a redução da regra e continua o processo recursivamente para a produção seguinte.

Depois de encontrar o primeiro *token* de sincronização, o analisador não assume que já se recuperou do erro. Em vez disso, ele entra em modo provisório. Nesse modo, o analisador tenta

empilhar até o terceiro *token* esperado. Se o analisador falhar durante o modo provisório, então ele retorna ao modo pânico desde o início de α, mas no mesmo lugar na cadeia de entrada.

O Yacc fornece algumas funcionalidades para tornar esse processo mais customizável. Ao inserir {yyerrok;} após um token de α, o analisador sintático entende que, depois desse token, o erro já foi resolvido, evitando que necessite empilhar três tokens para assumir que se recuperou do erro. Outra fucionalidade interessante é inserir o comando {yyclearin;} após um símbolo de erro para consumir o token inesperado da cadeia antes de começar o modo pânico, pois em algumas situações pode não ser desejável sincronizar com o token que causou o erro.

3.2 Escolha dos tokens de erro

Levando em consideração o funcionamento do símbolo de erro, é necessário evitar que haja conflitos na tabela sintática. Em outras palavras, um estado não pode ter duas ações diferentes para uma mesma transição, o que caracteriza uma ambiguidade na gramática. Essa não é uma tarefa fácil pois não é possível saber diretamente os estados da análise sintática apenas visualizando a gramática, e além disso um símbolo de erro na gramática pode representar um grande conjunto de possíveis *tokens* inesperados. Portanto para cada produção é necessário compreender bem em que situações essa produção pode ser chamada.

Antes de testar em quais símbolos de uma produção era possível se recuperar de um erro sem ambiguidade, primeiro nós tentávamos compreender a regra gramatical daquela produção, para identificar intuitivamente em quais casos era possível se recuperar de um erro e em quais casos não deveria ser possível se recuperar.

Depois testávamos cada possibilidade de substituição de símbolos, primeiro trocando símbolos terminais por símbolos de erro e depois trocando os símbolos não terminais. Se, ao compilar a gramática, houvesse um conflito do tipo empilhar-reduzir ou reduzir-reduzir, então consultávamos o arquivo de descrição do automato gerado para entender quais itens LR(0) estão em conflito e decidir quais regras de recuperação de erro manter.

A prioridade era manter produções de recuperação onde o símbolo de erro substituía um símbolo terminal, pois: eram erros mais fáceis de compreender por estarem nas folhas da árvore de derivação da linguagem; e simples de resolver por ser necessário apenas inserir um símbolo terminal faltante onde o erro ocorreu. Se o conflito ocorria entre produções de não terminais diferentes então a prioridade era manter produções mais simples, sem recursividade.

Por fim, não fizemos uso do comando yyerrok, então nosso analisador sintático só assume que se recuperou de um erro depois de empilhar três *tokens* sucessivamente, de maneira que erros que ocorram em menos de três *tokens* de distância serão assumidos como um único erro.

Referências

[1] Westes. Documentação do Flex. https://westes.github.io/flex/manual/index.html. Acessado em: 17/06/2019.

[2] GNU. Documentação do Bison. https://www.gnu.org/software/bison/manual/bison.html#Rpcalc-Declarations. Acessado em: 17/06/2019.

[3] Panic-mode error recovery. http://www.cs.ecu.edu/karl/5220/spr16/Notes/Bison/error.html. (Accessed on 06/17/2019).