Trabalho Prático 3
Compilador para a Linguagem P
Parte 2 - Análise Sintática

Arthur Souto Lima
Vitória Mirella Pereira do Nascimento

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

arthursl@ufmg.br

vitoriamirella@ufmg.br

1. Introdução

O trabalho prático consiste em implementar um compilador completo para a linguagem P descrita na documentação. Essa tarefa deve ser feita em partes, sendo a segunda delas implementar um analisador sintático com auxílio do Yacc e do Lex. Esse programa deve ser capaz de receber um programa fonte na linguagem P e verificar se está de acordo com a gramática definida na especificação. Nesse processo, são impressos na tela os tokens e algumas informações básicas acerca de cada um. Ao final, é informado de ocorreu um erro ou se a análise foi feita com sucesso, além da impressão da tabela de símbolos.

2. Instruções

2.1. Compilação

O analisador sintático foi desenvolvido usando o Yacc (do pacote GNU Bison), ao passo que o analisador léxico foi feito usando o Lex (no caso o Flex) e as funções auxiliares, bem como a tabela de símbolos, foram implementados em C. O programa acompanha um makefile para compilação e execução. Basta executar o comando *make* para compilar o programa. O executável final se chama *yysint* e estará na pasta raiz do programa (onde está também o makefile).

O makefile, nessa ordem, compila o módulo da tabela de símbolos, compila o analisador sintático por meio do Yacc, o módulo de funções auxiliares, o analisador léxico por meio do Lex e, finalmente, usando o scanner do analisador léxico (arquivo *lex.yy.c*) e o analisador sintático (arquivo *y.tab.c*) para compilar o programa que verificará o arquivo de entrada. Ato contínuo, ele já executa o arquivo de teste incluído.

2.2. Execução

Compilado o analisador sintático, ou seja, já foi gerado o executável *yysint*, deve-se rodá-lo junto com o código-fonte, no terminal, via o comando, por exemplo,

./yysint < teste1.in

Assim, o yysint chamará a função *yyparse* do Yacc que automaticamente já faz a descoberta dos tokens, já que o Yacc e o Lex estão muito bem integrados entre si e, em seguida, faz a análise sintática. Na tela serão impressos os tokens da análise léxica e sua

linha correspondente, algumas informações de inserção ou busca de elementos na tabela de símbolos, bem como o resultado final da análise sintática. Além disso, é impressa a tabela em si ao em seu estado final.

3. Implementação

3.1. Resumo do Projeto

O analisador sintático foi implementado com base nas aulas acerca do assunto, bem como das seções relevantes do livro-texto [Aho et al. 2007] e também no tutorial [Niemann]. Assim, a criação do arquivo fonte do Yacc foi bem direta, acompanhada de uma adaptação do analisador léxico criado na etapa anterior. Um *main* foi necessário para podermos ler o programa fonte e este foi adaptado dos exemplos de analisador sintático de [Niemann]. Esse *main* está no arquivo *yysint.y*.

A gramática foi implementada incrementalmente para que fosse possível testar as funcionalidades e solucionar os eventuais conflitos. Com o desenvolver do projeto, o arquivo de definição do analisador sintático ficou cada vez mais complexo e passou a representar corretamente a gramática passada.

A última parte desta etapa foi implementar a tabela de símbolos, que foi advinda da disponível na página da disciplina e encontra-se no módulo *tab*. Funções auxiliares da fase de análise léxica foram adaptadas durante o desenvolvimento para nova realidade, ainda no módulo *aux*. Essas funções dizem respeito sobretudo à análise léxica de identificadores e palavras reservadas, que apenas foram adaptadas para funcionar em conjunto com o Yacc.

3.2. Questões de Etapas Anteriores

Na fase de análise léxica, foram citadas algumas questões as quais, naquele momento, permaneciam em aberto, mas que, agora, após a conclusão do analisador sintático, foram resolvidas total ou parcialmente.

Em primeiro lugar, o analisador sintático, por ser sua função, encaixa a entrada e os tokens formados pelo analisador léxico nas produções da gramática, ação que a etapa anterior não era capaz de fazer. Desse modo, é possível reconhecer estruturas mais complexas do programa. Além disso, por conhecer essas regras, ele consegue resolver a questão do menos unário (mais detalhes na subseção 3.5.3).

Ainda nessa etapa, foi implementada a tabela de símbolos, além de que foi iniciado também seu preenchimento, apesar de não ser definitivo e estar sujeito a alterações futuras nas etapas posteriores.

3.3. Questões não solucionadas

Apesar de consideráveis avanços, o compilador, ao fim desta etapa ainda não está completo. Toda a parte de análise semântica, verificação de tipos, avaliação de expressões e geração de código intermediário ainda segue em aberto. Ademais, por não haver essa ponderação semântica, a tabela de símbolos, até o momento é apenas escrita, isto é, não utilizamos os dados ali guardados para qualquer ação.

Como comentado, o analisador sintático não consegue avaliar totalmente os tipos de expressões, o que leva a alguns avisos do Yacc sobre eventuais conflitos de tipos quando das reduções

3.4. Suposições e Modificações Feitas

Tal como na etapa anterior, passou-se a regra gramatical que identificava o que era uma constante booleana para as definições léxicas, já que todas as outras constantes ali estavam definidas. Desse modo, o analisador sintático recebe diretamente o token de constante booleana do analisador léxico.

Além disso, também foi feita uma modificação na produção

```
simple_variable_or_proc ::= identifier
```

Para

```
simple_variable_or_proc ::= identifier_f
                          | eoln
```

Em que o token *identifier_f* representa alguma função padrão definida na especificação. Apesar de a primeira forma não gerar conflitos, com a segunda conseguimos restringir que as funções padrão sejam consideradas palavras reservadas, ou seja, se colocadas no programa de entrada, será considerada como uma função e não como um identificador comum. Isso causará um erro de sintaxe caso alguma delas seja utilizada de forma equivocada.

O *eoln* também está ali pois, como descrito na subseção 3.5.1, há um conflito e, na sua solução, elicitamos o token referente à função padrão *eoln*. Convém lembrar que ela pode ser usada com ou sem parâmetros, seguindo o funcionamento de Pascal. Assim, a mudança escrita na subseção 3.5.1 representa o uso sem parâmetros, enquanto a mudança descrita nesta seção equivale a sua invocação com parâmetros.

3.5. Conflitos

A gramática possui dois conflitos principais.

3.5.1. Conflito Reduce-Reduce de Identificador

O primeiro deles é um conflito reduce-reduce de identificador nas produções:

```
function_ref ::= identifier
factor := identifier
```

O compilador não conseque saber quando reduzir a um ou a outro. Para solucioná-lo, foi feita a seguinte modificação:

```
function_ref ::= EOLN
factor := identifier
```

Sendo que "*EOLN*" é o token da função padrão EOLN, definida na especificação e com comportamento semelhante ao da linguagem Pascal. Isso vale pois essa função é a única, dentre as listadas na documentação, que pode ser chamada sem argumentos, que, no caso, deve retornar *true* quando a entrada padrão chegar ao fim de linha.

Além disso, ainda sobre esse conflito, a função EOLN é a única cujo token retornado pelo analisador léxico não é *identifier_f*, fazendo referência a modificação apresentada na subseção 3.4.

3.5.2. Conflito Shift-Reduce If-Then-Else

O outro conflito que existe na gramática dada é um conflito shift-reduce devido às produções:

```
if_stmt ::= if cond then stmt
          | if cond then stmt else stmt
```

Tal como comentado em [Niemann] e também em [Aho et al. 2007], podemos usar regras de associatividade e de precedência para resolver alguns conflitos. A solução adotada foi a mesma empregada em [Niemann], definimos precedência:

```
if_stmt :   IF cond THEN stmt   %prec THEN
        |   IF cond THEN stmt ELSE stmt
```

E não-associatividade desses tokens no arquivo de definição da gramática *yysint.y*.

```
%nonassoc THEN
%nonassoc ELSE
```

Isso resolve o problema do chamado "dangling-else". Apesar das regras padrão do Yacc preferirem o shift ao reduce, foi implementada a solução desse conflito para remover o aviso e deixar explícita a intenção do projetista.

3.5.3. O "Menos Unário"

Em aula, vimos um exemplo no qual conseguimos utilizar uma gramática ambígua no Yacc resolvendo a ambiguidade do menos unário ("*unary minus*"). Contudo, no caso da gramática da especificação, essa estratégia não foi necessária já que o menos unário está numa produção diferente da subtração o que evita esse problema e faz com que o analisador sintático corretamente diferencie o menos unário do sinal de subtração sem a intervenção explícita do projetista.

3.6. Tabela de Símbolos

A tabela de símbolos utilizada foi adaptada da disponível na página da disciplina. Escolheu-se a implementação da floresta de árvores binárias, dada suas vantagens nos mais diversos aspectos, como visto em aula.

A única adaptação feita foi quanto aos atributos que a tabela guardaria. No caso, adicionamos dois campos ao *struct* da entrada da tabela, substituindo o campo único *atributo* que existia originalmente. O novo campo *type*, como o nome denota, guarda o tipo do identificador, enquanto o campo *value* guarda o valor daquele identificador. O campo do tipo já está sendo corretamente utilizado pelo compilador, contudo o campo do valor ainda não, já que o compilador ainda não consegue avaliar os valores das expressões para corretamente guardar um valor na tabela.

Uma outra observação é que guarda-se o valor numa string. Isso foi uma decisão de implementação no momento. Posteriormente, na fase de análise semântica e de geração de código intermediário, será possível realmente usar os valores e avaliar as expressões. Chegando lá, transformaremos-se essa string numa constante real, inteira ou outra constante dentre as definidas na especificação. Naturalmente, essa decisão de projeto não é definitiva, e pode ser que, com o avançar do trabalho prático, ela seja revisada e uma outra estratégia seja escolhida.

Finalmente, convém ressaltar que foi criada uma função para atualizar o campo *value* de uma entrada. Novamente, essa funcionalidade foi implementada agora mas com pensamento já nas etapas posteriores, quando o compilador já será capaz de avaliar as expressões e corretamente preencher esses campos na tabela de símbolos. Nesse momento, ela é utilizada, para fins de teste apenas, colocando o valor de um contador sempre que há uma atribuição.

4. Exemplos e Testes

Diferentemente do que foi feito na documentação do analisador léxico, não conseguiremos reproduzir o "hardcopy" do terminal por completo, dada a sua extensão. Contudo, os testes foram fornecidos em conjunto com código-fonte do projeto para que possam ser executados como descrito nas instruções e sua saída seja visualizada.

São fornecidos dois testes, um com um erro de sintaxe e um bem parecido, mas que está de acordo com a gramática da linguagem P. Tal como na primeira etapa do projeto, reiterando o que foi descrito nas instruções, o segundo teste é automaticamente executado pelo makefile após a compilação do programa, ou seja, após o comando *make*.

4.1. Teste 1: erro de sintaxe

No arquivo *teste1.in* temos um programa, mas com um erro de sintaxe sutil. Na última declaração, não deveria haver um ";", pelas definições da gramática.

```
program mytest1;

x, k, i: integer;
y: real;
z: integer;

begin
    ...
end
```

Corretamente, o analisador sintático apresenta um erro:

```
./yysint < teste1.in
[   1] Reserved:        program
[   1] Identifier:      mytest1
[   3] Identifier:      x
[   3] Identifier:      k
[   3] Identifier:      i
[   3] Reserved:        integer
3 declaracoes do tipo integer
Instalando x, do tipo integer
Instalando k, do tipo integer
Instalando i, do tipo integer
[   4] Identifier:      y
[   4] Reserved:        real
1 declaracoes do tipo real
Instalando y, do tipo real
[   5] Identifier:      z
[   5] Reserved:        integer
1 declaracoes do tipo integer
Instalando z, do tipo integer
[   7] Reserved:        begin
parse error
```

4.2. Teste 2: programa correto

A seguir, no arquivo *teste2.in*, temos um programa muito parecido com o do teste anterior, contudo está com o erro supracitado corrigido.

```
program mytest2;

x, k, i: integer;
y: real;
z: integer

begin
    ...
    begin
        ...
        goto label2
    end;
    z := cos(45)
end
```

Como o programa está correto, o analisador sintático confirma essa corretude e, ao final, imprime o estado final da tabela de símbolos.

```
./yysint < teste2.in
[   1] Reserved:        program
[   1] Identifier:      mytest2
[   3] Identifier:      x
[   3] Identifier:      k
[   3] Identifier:      i
[   3] Reserved:        integer
3 declaracoes do tipo integer
Instalando x, do tipo integer
Instalando k, do tipo integer
Instalando i, do tipo integer
[   4] Identifier:      y
...
[  42] Reserved:        goto
[  42] Identifier:      label2
[  43] Reserved:        end
[  44] Reserved:        end
[  45] Identifier:      z
[  45] Assign:  :=
[  45] Function:        cos
[  45] Integer:         35
[  46] Reserved:        end
Encontrado z. O item está no nível: 1, indice: 5
Parse sucessful


Nome : x
Tipo : integer
Valor : 20
Nivel : 1
Esquerdo : 0
Direito : 2
...
```

5. Conclusão

Ao final dessa segunda parte deste trabalho prático, novamente pôde-se colocar em prática os conhecimentos adquiridos nas aulas acerca da construção de um analisador por intermédio de uma ferramenta, no caso um analisador sintático via o Yacc. É importante reiterar, mais uma vez, a ímpar possibilidade de experienciar as duas situações: implementar um *parser* feito sem tal ferramenta, como foi no Trabalho Prático 2, e, agora, utilizar uma ferramenta para automatizar uma boa parte dessa tarefa numa gramática consideravelmente maior. Convém ressaltar, ainda, a facilidade de implementar a tabela de símbolos, visto que os códigos das estruturas vistas em aula estavam disponíveis na página da disciplina e bastou adaptá-los para a realidade do projeto. Por fim, é evidente, no contexto de todo o compilador da linguagem P que deve ser feito, que ainda há um caminho a ser trilhado até o fim.

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. (2007). *Compilers Principles,Techniques, Tools*. Addison Wesley.

Niemann, T. Lex & yacc tutorial. [1]. Acesso em: 07 fev 2021.

[1] https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf