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. Dependências

O projeto possui duas dependências externas o Flex e o GNU Bison, para, respectivamente, o Lex e o Yacc. Cada um destes pode ser instalado usando o gerenciador de pacotes, por exemplo, do Ubuntu:

```
sudo apt-get install flex bison
```

Os dois estarão instalados corretamente e prontos para executar o programa se os comandos a seguir funcionarem

```
lex --version
yacc --version
```

No caso do desenvolvimento do projeto, a versão do Flex utilizada é a 2.6.4 (de 2017) e a versão do GNU Bison é a 3.5.1 (de 2020)

2.2. Compilação

Para compilar o programa, deve-se utilizar o makefile fornecido, por meio do comando *make*. Os módulos auxiliares serão automaticamente compilados pelo GCC e o analisador léxico estará disponível no executável *yysint*.

2.3. Execução

Para executar o programa já compilado, deve-se redirecionar um arquivo de texto para a entrada-padrão, usando o caractere "menor que" ("<"), no caso[1]. Por exemplo, para analisar um programa na linguagem P que esteja no arquivo *teste1.in*, deve-se executar

```
./yysint < teste1.in
```

2.4. Testes

Para executar os testes apresentados nessa documentação, há duas opções. A primeira é utilizar o makefile, via o comando

```
make tests
```

Assim, o makefile executará o programa para os testes fornecidos *teste0.in*, *teste1.in* e *teste2.in* segundo o padrão descrito na seção anterior.

A outra opção é executar manualmente cada teste, como foi feito na seção citada acima. Desse modo, para executar o *teste1.in*, por exemplo, deve-se usar:

```
./yysint < teste1.in
```

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.

O analisador sintático foi desenvolvido usando o Yacc (do pacote GNU Bison), ao passo que o analisador léxico foi feito usando o Lex (do pacote Flex) e as funções auxiliares, bem como a tabela de símbolos, foram implementados em C. Para amparar a compilação e execução, o programa acompanha um makefile. Instruções de compilação e execução estão nas subseções relevantes da seção 2. O executável final se chama *yysint* e estará na pasta raiz do programa, onde está também o makefile. O analisador léxico está definido no arquivo *yylex.l* e o analisador sintático no arquivo *yysint.y*.

Durante a execução, *main* 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 será impresso o resultado da análise sintática, bem como o estado final da tabela de símbolos.

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.

[1]Mais informações em: `https://tldp.org/LDP/abs/html/io-redirection.html`

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.

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 seguem 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.

3.4. Suposições e Modificações Feitas

3.4.1. Constantes Booleanas

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.

3.4.2. Sinal de Constantes Numéricas

A gramática da especificação não define que um sinal esteja junto de uma constante. Contudo, seguindo as orientações da professora e o modelo da gramática do TP2, foram adicionadas as seguintes definições léxicas na parte de constantes:

```
int     ::=     sign unsigned_integer
real    ::=     sign unsigned_real
```

Isso permite que operadores unários de sinal (”+” e ”-”) sejam utilizados pelo analisador léxico para reconhecer constantes com sinal. Em contrapartida, isso exige que esses sinais de constantes sempre estejam ”colados” à constante, isto é, não haja espaço entre ele e a constante.

3.4.3. Questão do *funtion_ref*

Tal como no TP2, com uma gramática muito similar, foi feita uma modificação na produção de chamada de função. Assim, a produção

```
function ref ::= identifier
               |  function_ref_par
```

foi modificada para apenas

```
function ref ::= function_ref_par
```

Ainda nesse contexto, para tornar as funções padrão palavras reservadas, modificou-se a produção seguinte.

```
simple_variable_or_proc ::= identifier
```

Para

```
simple_variable_or_proc ::= identifier_f
```

Em que o token *identifier_f* representa alguma função padrão definida na especificação. Apesar de a forma original dessa produção *simple_variable_or_proc* não gerar conflitos, como comentado, 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.

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, fizemos as modificações descritas na subseção 3.4.3, seguindo a solução adotada no TP2.

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.

3.6. 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

Os testes a seguir estão disponíveis também em conjunto com o código-fonte do programa. Tal como na primeira parte do projeto, apresentaremos a entrada e o "hardcopy" do terminal com a saída.

São fornecidos três testes, o primeiro (*teste0.in*) é um teste de um programa bem simples na linguagem P. O segundo e o terceiro são mais complexos e parecidos entre si, o *teste1.in* está correto e o *teste2.in* possui um erro de sintaxe. Mais informações sobre como executar os testes aqui citados podem ser encontradas na subseção 2.4.

4.1. Teste 0: programa simples em P

No arquivo *teste0.in*, temos o simples programa a seguir, implementado na linguagem P.

```
program mytest0;

x: integer

begin
    x := 1 + 1;
    x := x + 2
end
```

A saída, demonstrando a corretude da sintaxe do programa, juntamente com a impressão da tabela de símbolos está abaixo. Lembrando, como comentado, o valor guardado na tabela de símbolos é apenas um contador, pois o programa não consegue avaliar expressões ainda.

```
./yysint < teste0.in
Parsing...
Parse sucessful

Tabela de Simbolos Final:

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

4.2. Teste 1: programa correto em P

No arquivo *teste1.in* temos um programa consideravelmente mais complexo, com produções mais elaboradas, mas todas corretas.

A saída do programa é colocada abaixo. Não foi apresentada toda a tabela de símbolos aqui, apenas parte, para não ficar muito extenso.

```
./yysint < teste1.in
Parsing...
Parse sucessful

Tabela de Simbolos Final:

Nome : x
Tipo : integer
Valor : 11
Nivel : 1
Esquerdo : 0
Direito : 2

...
```

```
program mytest1;

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

begin
    begin
        x := sin(30);
        y := (10 * -3.5E10 + -(5*7));
        z := 0;
        label2: while (x <= +4.0002) do
            z := z + y;
            x := x + 1
        end;

        do
            z := z-sin(x);
            k := x - +1E+1;
            while (NOT (x <= 5)) do
                z := z + y;
                x := x + 1
            end
        until (x <= 0);

        do
            z := z-x;
            x := x + -1
        until (x <= 0)
    end;
    begin
        goto label1;
        begin
            label1: read (file3);
            write (1 + 1156.07257E-10,teste2);
            if NOT eoln(file3) then
                y := 1
            else
                y := 2
            ;
            goto label2
        end
    end;
    z := cos(35)
end
```

Arquivo *teste1.in*

4.3. Teste 2: programa com erro

Finalmente, o *teste2.in* é basicamente o mesmo programa do *teste1.in*, contudo ele possui um erro de sintaxe. No caso, esse erro é um ponto-e-vírgula equivocado após a última declaração.

```
program mytest2;

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

begin
    /* Idêntico ao teste1.in */
end
```

Ao detectar o erro de sintaxe, o analisador simplesmente acusa o erro e encerra a sua execução.

```
./yysint < teste2.in
Parsing...
syntax error
```

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. [2]. Acesso em: 07 fev 2021.

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