



Universidade Federal da Fronteira Sul - UFFS
Campus de Chapecó
Curso de Ciência da Computação

Jod F Pierre

Disciplina: Construção de compiladores

Chapecó
2021

Jod F Pierre

Projeto e implementação das etapas de compilação para uma linguagem de programação hipotética apresentado ao curso de ciência da computação da Universidade Federal da Fronteira Sul, como requisito parcial para aprovação na disciplina de Construção de compiladores ministrada pelo professor: Bráulio Adriano De Mello

**Chapecó
2021**

RESUMO

Este trabalho mostra as duas principais etapas de um compilador que, a partir de um arquivo fonte contendo um conjunto de tokens e gramática livre de contexto, consegue construir o autômato finito determinístico. Este autômato é usado em seguida para identificar o estado de cada token reconhecido no analisador léxico que é a primeira fase de um compilador. Após a identificação dos tokens, é realizada a segunda etapa, que é o analisador sintático onde, a partir de um conjunto de regras definidas previamente, o analisador consegue dizer se um determinado texto é reconhecido ou não sintaticamente pela gramática. Para isto, foi usado a ferramenta goldparser que ajuda na construção da tabela LALR contendo os símbolos, as produções e os estados.

INTRODUÇÃO

Um compilador é um sistema escrito por programadores que aceita como entrada um programa escrito em uma linguagem de programação e produz como saída um outro programa equivalente em outra linguagem. Para produzir o código equivalente a partir do código fonte recebido, o compilador analisa a entrada passo a passo até concluir todos. Antes de iniciar um novo passo, certifique-se de não ter nenhum erro. Caso tenha, lança mensagens de erros informando onde está exatamente e o que o ocasionou. Assim, se não ocorrer nenhum erro, o código equivalente será produzido. Neste trabalho, foi implementado as duas(2) primeiras fases do compilador que são sucessivamente os analisadores léxico e sintático. Para o analisador léxico, foi usado o trabalho da disciplina de Linguagens formais e autômatos(LFA) de terceiro adaptado conforme a necessidade para gerar o autômato finito determinístico que foi usado no analisador léxico para identificar o estado dos tokens reconhecidos. No analisador sintático, foi usado a ferramenta goldparser que facilitou o processo de criação da tabela LALR que é a raiz do analisador sintático.

REFERENCIAL TEÓRICO

Analisador léxico

Identificar sequências de caracteres que constituem a unidades léxicas(tokens). No analisador léxico, é feita a leitura do texto fonte caractere

a caractere para verificar se o determinado token pertence à linguagem previamente definida. Por exemplo, se na linguagem foi definido que para ser uma variável, deve iniciar com o símbolo `_` e deve conter somente letras minúsculas de a a d(Exemplo: `_abc`). Se violar um dos critérios, ao declarar uma variável, o compilador irá disparar um erro léxico.

Analizador sintático

Se não tiver nenhum erro léxico, o analisador sintático é responsável por analisar o conteúdo recebido do analisador léxico junto ao código fonte a ser analisado. Para isso, é usado o goldparser que é uma ferramenta disponível para rodar no sistema operacional windows. Com ela, ao invés de criar a tabela manualmente, basta inserir a gramática sintática livre de conflito, que irá gerar a tabela LALR, os símbolos e os possíveis estados para prosseguir com o reconhecimento sintático.

IMPLEMENTAÇÃO

Para implementação das etapas do compilador, foi usado a linguagem python por fins de aprimoramento na linguagem, também por proporcionar uma certa facilidade por ser uma linguagem fácil com várias bibliotecas disponíveis para uso e por ter uma comunidade muito ativa.

Geração de autômato

A geração do autômato é realizada a partir do código fonte de entrada contendo o conjunto de tokens e gramática aceita pela linguagem.

`./materials/token_and_GR.txt`

```

materials > ≡ token_and_GR.txt
1  |if
2  else
3  print
4  while
5  +
6  -
7  *
8  /
9  +:
10 +:=
11 :-
12 :-=
13 =
14 ==
15 (
16 )
17 !=
18 !
19 {
20 }
21
22 <S> ::= _<A>
23 <A> ::= a<A> | b<A> | c<A> | d<A>
24 <B> ::= a<B> | b<B> | c<B> | d<B> | $
25
26 <S> ::= 1<A> | 2<A> | 3<A> | 4<A>
27 <A> ::= $
28

```

Pela imagem acima, no analisador léxico, será aceito somente os tokens que estão nesse conjunto de tokens ou aqueles que atendem às regras da gramática. Ou seja, pela primeira gramática, só aceita palavra iniciando com o símbolo underline(_) e tal palavra deve conter somente as seguintes letras: a, b, c, d. Exemplos de entrada válida: _b, _abc. Pela segunda gramática, é aceito número de tamanho único de 1 a 4. Baseando nisso, será gerado o autômato finito determinístico com os estados dos tokens ou palavras reconhecidas.

Analizador léxico

É a primeira fase do compilador onde é feita a leitura de um código fonte caractere a caractere para verificar se atende às solicitações do conjunto de tokens

ou das gramáticas definidas na geração do autômato. Se o token é inválido, será lançada mensagem de erro informando qual token que a provocou, qual linha e coluna. Se o token foi reconhecido pelo analisador léxico, o mesmo será adicionado numa tabela de símbolos junto com as suas demais informações(linha, coluna, estado, rótulo e tipo). Abaixo, são alguns exemplos de código de entrada.

```
materials > ≡ code_examples.txt
1 | #handle lexical error(input line by line)
2 | _a = b
3 | _a = _i
4 | _a = 5
5 |
6 | #Accepted(input line by line)
7 | _a = _b
8 | _a = 2
9 | print ( _d )
10 |
11 | #Accepted
12 | while _a = _b {
13 | | _a = 2
14 | }
15 |
16 | #Accepted
17 | if _a += 2 {
18 | | print ( _a )
19 | } else {
20 | | _a = _a + 1
21 | }
```

./materials/input_code.txt

Itens da tabela de símbolo do analisador léxico para a entrada: **_a = 2**

```
λ jodfedlet src → λ git main* → python3 main.py
{'Line': 1, 'Column': 1, 'State': 44, 'Label': '_a', 'Type': 'ID'}
{'Line': 1, 'Column': 2, 'State': 51, 'Label': '=', 'Type': 'RELAT(EQUAL)'}
{'Line': 1, 'Column': 3, 'State': 44, 'Label': '2', 'Type': 'Number'}
λ jodfedlet src → λ git main* →
```

Mensagem de erro para a entrada: **_a = 5**

```
materials > ≡ input_code.txt
1 | _a = 5|
```

```
λ jodfedlet src → λ git main* → python3 main.py
(LexicalError) --> Token 5 at position: file[1, 3] is not valid
λ jodfedlet src → λ git main* →
```

Analizador sintático.

Ao receber o conjunto de tokens reconhecidos pelo analisador léxico, o analisador sintático é responsável por verificar se as expressões se encaixam dentro das regras sintáticas definidas.

Regras sintáticas: ./materials/syntax_GR.txt

```
materials > ≡ syntax_GR.txt
1 "Start Symbol" = <S>
2 <S> ::= <ID> '=' <EXPR> | <CONDITIONAL> | <LOOP> | 'print' '(' <ID> ')'
3 <CONDITIONAL> ::= 'if' <EXPR> '{' <S> '}' <CONDITIONALP>
4 <CONDITIONALP> ::= 'else' '{' <S> '}' | <>
5 <LOOP> ::= 'while' <EXPR> '{' <S> '}'
6 <OP> ::= '+' | '-' | '*' | '/' | '=' | '+=' | '+:=' | ':=' | '-=' | '!=' | '=='
7 <EXPR> ::= <ID> | <ID> <OP> <ID> | <ID> <OP> <NUMBER> | <NUMBER> <OP> <NUMBER> | <NUMBER> <OP> <ID> | <NUMBER>
8 <NUMBER> ::= 1<NUM> | 2<NUM> | 3<NUM> | 4<NUM>
9 <NUM> ::= <>
10 <ID> ::= '_ID'
11
```

A partir dessa gramática gerada, foi usada a ferramenta goldparser para gerar um arquivo XML (./materials/parser.xml) contendo os símbolos, as produções e a tabela LALR(Look Ahead LR).

Desse arquivo, é feito o carregamento dos símbolos, produções e a tabela LALR. Depois disso, é feito o mapeamento da tabela de símbolos baseando-se nas regras sintáticas. Por exemplo, ao encontrar uma palavra iniciando com o underline(_), a última regra sintática nos diz que é uma variável. Desse modo, foi feita a substituição do estado do token da tabela pelo estado do símbolo identificado pelo parser. Por fim, foi gerada a fita contendo os estados dos tokens.

A tabela de símbolos contém as ações a serem usadas no processo de reconhecimento. Dentro delas, usamos a 1 que é pra empilhar, a 2 que é pra reduzir e a 4 que é pra aceitar a palavra.

Para iniciar o processo de reconhecimento sintático, temos uma pilha(inicialmente com valor 0) e a fita recebida depois do mapeamento que tem os estados do texto a ser analisado. Em seguida, a ação atual é buscada na tabela LALR com o valor inicial da pilha e o estado do token a ser analisado: `action = tabela_lalr[stack[0]][ribbon[0]]`. Se não conseguir encontrar a ação, é porque ocorreu um erro sintático, então é lançada uma mensagem de erro sintático informando qual token que o provocou. Por exemplo, na primeira regra sintática, para imprimir algo na tela, deve ser nesse formato: `'print' '(' <ID> ')'` onde `<ID>` é toda palavra que inicia com `_`. Então, ao usar o formato `print (_a` irá disparar o seguinte erro:

```
λ jodfedlet src → λ git main* → python3 main.py
(SyntaxError) --> Token _a at position: file[1, 3] is not valid
λ jodfedlet src → λ git main* →
```

Isso porque foi informado o fechamento dos parênteses.

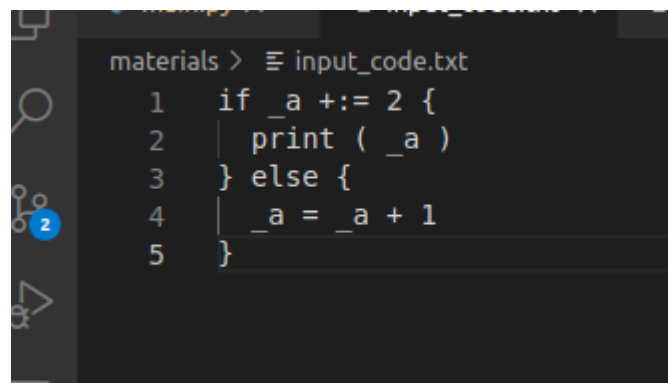
Caso a ação foi detectada corretamente:

Se for 1(empilhamento), foi adicionado o primeiro estado da fita no início da pilha junto ao valor, depois a fita foi movida para a próxima posição. Ou seja, foi removido o valor da primeira posição.

Se for 2(Redução), buscamos a produção do arquivo parser a partir do valor da ação. Ocorreu desempilhamento até 2 vezes do tamanho do símbolo da produção. Depois disso, o não terminal entra na pilha junto ao valor.

Se a ação for 4, o analisador reconhece a sentença como válida.

Exemplo:



```
materials > input_code.txt
1  if _a += 2 {
2    print ( _a )
3  } else {
4    _a = _a + 1
5  }
```



```
λ jodfedlet src → λ git main* → python3 main.py  
OK -> Accepted  
λ jodfedlet src → λ git main* →
```

CONCLUSÃO

Com este trabalho realizado, dá pra entender melhor o trabalho gigantesco realizado pelo compilador que é analisar passo a passo o código de entrada(sentença) para depois decidir se é válido ou não, baseado nos critérios definidos pelas gramáticas. Umas dificuldades que foram encontradas são referente à manipulação do arquivo do parser por não ter muito material disponível referente à ferramenta.

REFERÊNCIAS

- <http://www.ybadoo.com.br/tutoriais/cmp/>
- <http://www.goldparser.org/>
- <https://www.youtube.com/watch?v=8rB8Dvczc1g&list=PL0Z-gyL9saMcajYH26KWKQG0nH2C2fsMQ>
- <https://www.youtube.com/watch?v=EEKkSsHEKyg&t=1885s>

Código fonte: <https://github.com/jodfedlet/Compiler-UFFS>