Alan D. Barroso, Kenji Sakata Jr

Anlisador Sintático

Universidade de São Paulo
Escola Politécnica

Linguagens e Compiladores
PCS2056

São Paulo
2013

1 Introdução

O analisador sintático é o responsável por reconhecer a estrutura da linguagem, formada a partir da gramática. Sua ação começa uma vez que o módulo léxico já reconheceu todas as partículas átomicas que compõem o código-fonte.

Na realidade, é o analisador sintático quem invoca a ação do módulo léxico em um primeiro momento a fim de identificar as partículas do texto. Em seguida, ele, o analisador sintático, faz uma segunda leitura do código-fonte, mas, desta vez, ele tenta descrever como essas partículas estão estruturadas e organizadas no texto. A estrutura que é composta por esta segunda leitura é chamada de árvore de sintaxe e consiste unicamente em uma forma alternativa de representar o código-fonte, mas a qual é compreensível pelo compilador.

Em seguida, quando da análise semântica, é novamente o módulo sintático quem será o responsável por invocar o analisador semântico, o qual traduzirá a árvore de sintaxe em ações concretas, executáveis pelo computador.

Outras funções tipicamente atribuídas ao analisador sintático são detecção de erros de sintaxe, recuperação de erros, correção de erros, ativação de rotinas de síntese do código-objeto, entre outras.

Existem outras formas de delegar essas tarefas aos três módulos canônicos de um compilador clássico, mas podemos concluir que, por conta de nossas decisões de projeto, o compilador em desenvolvido é orientado à sintaxe. Isso porque o módulo sintático do compilador é o pivô central que coordena o fluxo sequencial de ações que permitem traduzir uma linguagem de alto nível em linguagem de máquina. É ele quem determina qual é a etapa do processo de compilação que está sendo executada e quem é o responsável por ela.

Assim, após definir a linguagem e desenvolver o analisador léxico, tratamos de construir o módulo de análise sintática, cujo método de reconhecimento foi baseado em autômato de pilha estruturado (APE).

2 Geração Automática dos Autômatos

2.1 Notação de Wirth Reduzida

Na primeira etapa para a criação do APE, foram utilizadas as expressões fundamentais criadas a partir da descrição reduzida em notação de Wirth. A definição das expressões seguiu o critério dos não-terminais com recursividade central, sendo possível concluir que as máquinas finais tratariam a sequência de comandos, as expressões e o programa principal.

Assim, a linguagem representada pela notação de Wirth reduzida pode ser encontrada no arquivo "wirth_reduzido.txt"que acompanha esse documento.

2.2 Lista de submáquinas do APE

2.2.1 Lista de transições

A partir da linguagem representada em notação de Wirth reduzida, utilizamos o programa do site indicado [1] para gerar automaticamente a tabela de transições as quais as submáquinas deveriam executar de forma a caracterizar a linguagem.

O programa gera diversas tipos de saídas. Cuida lembrar que somente consideramos a saída que é uma tabela de transição reduzida, de forma a otimizar o processamento do compilador e facilitar a leitura e boa compreensão da representação dos autômatos, seja ela na forma tabular ou gráfica.

Grosso modo, o programa utiliza três etapas para reduzir a tabela de transições. A ordem em que elas devem ser executadas é a seguinte:

- Eliminação das transiçõees em vazio;
- Eliminação dos estados não-acessíveis;
- Eliminação dos estados equivalentes.

A saída do tabela referente à submáquina programa:

[1] http://radiant-fire-72.heroku.com/

```
initial: 0
final: 29
(0, "const") -> 1
(0, "typedef") -> 2
(0, "struct") -> 3
(0, "vars") -> 4
(0, "function") -> 5
(0, "program") -> 6
(1, identificador) -> 7
(2, identificador) -> 23
(3, identificador) -> 30
(4, identificador) -> 53
(4, "bool") -> 53
(4, "int") -> 53
(4, "char") -> 53
(4, "float") -> 53
(5, identificador) -> 31
(5, "bool") -> 31
(5, "int") -> 31
(5, "char") -> 31
(5, "float") -> 31
(5, "void") -> 31
(6, "begin") -> 8
(7, "=") -> 9
(8, "end") -> 10
(8, "vars") -> 11
(8, comando) -> 12
(9, inteiro) -> 13
(9, float) -> 13
(9, "true") -> 13
(9, "false") -> 13
(9, char) -> 13
(10, "program") -> 29
(11, identificador) -> 15
(11, "bool") -> 15
(11, "int") -> 15
(11, "char") -> 15
(11, "float") -> 15
(12, "end") -> 10
(12, comando) -> 12
(13, ";") -> 14
(14, identificador) -> 7
(14, "end") -> 16
(15, identificador) -> 17
(16, "const") -> 18
(17, ";") -> 19
(17, "[") -> 20
(17, ",") -> 15
(18, "typedef") -> 2
(18, "struct") -> 3
(18, "vars") -> 4
```

```
(18, "function") -> 5
(18, "program") -> 6
(19, identificador) -> 15
(19, "end") -> 22
(19, "bool") -> 15
(19, "int") -> 15
(19, "char") -> 15
(19, "float") -> 15
(20, inteiro) -> 21
(21, "]") -> 17
(22, "vars") -> 12
(23, "=") -> 24
(24, identificador) -> 25
(24, "bool") -> 25
(24, "int") -> 25
(24, "char") -> 25
(24, "float") -> 25
(25, ";") -> 26
(26, identificador) -> 23
(26, "end") -> 27
(27, "typedef") -> 28
(28, "struct") -> 3
(28, "vars") -> 4
(28, "function") -> 5
(28, "program") -> 6
(30, "begin") -> 32
(31, identificador) -> 33
(32, identificador) -> 34
(32, "bool") -> 34
(32, "int") -> 34
(32, "char") -> 34
(32, "float") -> 34
(33, "(") -> 35
(34, identificador) -> 36
(35, identificador) -> 37
(35, "bool") -> 37
(35, "int") -> 37
(35, "char") -> 37
(35, "float") -> 37
(35, ")") -> 38
(36, ";") -> 39
(36, "[") -> 40
(36, ",") -> 34
(37, identificador) -> 59
(37, "byref") -> 60
(38, "begin") -> 41
(39, identificador) -> 34
(39, "end") -> 51
(39, "bool") -> 34
(39, "int") -> 34
(39, "char") -> 34
```

```
(39, "float") -> 34
(40, inteiro) -> 45
(41, "end") -> 42
(41, "vars") -> 43
(41, comando) -> 44
(42, "function") -> 58
(43, identificador) -> 46
(43, "bool") -> 46
(43, "int") -> 46
(43, "char") -> 46
(43, "float") -> 46
(44, "end") -> 42
(44, comando) -> 44
(45, "]") -> 36
(46, identificador) -> 47
(47, ";") -> 48
(47, "[") -> 49
(47, ",") -> 46
(48, identificador) -> 46
(48, "end") -> 52
(48, "bool") -> 46
(48, "int") -> 46
(48, "char") -> 46
(48, "float") -> 46
(49, inteiro) -> 50
(50, "]") -> 47
(51, "struct") -> 28
(52, "vars") -> 44
(53, identificador) -> 54
(54, ";") -> 55
(54, "[") -> 56
(54, ",") -> 53
(55, identificador) -> 53
(55, "end") -> 63
(55, "bool") -> 53
(55, "int") -> 53
(55, "char") -> 53
(55, "float") -> 53
(56, inteiro) -> 57
(57, "]") -> 54
(58, "function") -> 5
(58, "program") -> 6
(59, "[") -> 61
(59, ",") -> 62
(59, ")") -> 38
(60, identificador) -> 59
(61, inteiro) -> 64
(62, identificador) -> 37
(62, "bool") -> 37
(62, "int") -> 37
(62, "char") -> 37
```

```
(62, "float") -> 37
(63, "vars") -> 58
(64, "]") -> 59
```

../1–linguagem/notacoes/JFLAP/programa/programa.txt

A saída referente à submáquina comando:

```
initial: 0
final: 12
(0, identificador) -> 1
(0, "call") -> 2
(0, "if") -> 3
(0, "while") -> 4
(0, "scan") -> 5
(0, "print") -> 6
(0, "return") -> 7
(1, "[") -> 8
(1, ".") -> 9
(1, "=") -> 10
(1, "+=") -> 10
(1, "-=") -> 10
(1, "*=") -> 10
(1, "/=") -> 10
(2, identificador) -> 18
(3, "(") -> 22
(4, "(") -> 28
(5, identificador) -> 14
(6, expressao) -> 13
(7, expressao) -> 11
(7, ";") -> 12
(8, expressao) -> 16
(9, identificador) -> 1
(10, expressao) -> 11
(11, ";") -> 12
(13, ";") -> 12
(13, ",") -> 6
(14, "[") -> 15
(14, ".") -> 5
(14, ";") -> 12
(14, ",") -> 5
(15, expressao) -> 17
(16, "]") -> 1
(17, "]") -> 14
(18, "(") -> 19
(19, expressao) -> 20
(19, ")") -> 11
(20, ",") -> 21
(20, ")") -> 11
(21, expressao) -> 20
(22, expressao) -> 23
(22, ")") -> 24
```

```
(23, ")") -> 24
(24, "then") -> 25
(25, comando) -> 25
(25, "else") -> 26
(25, "end") -> 27
(26, comando) -> 26
(26, "end") -> 27
(27, "if") -> 12
(28, expressao) -> 29
(28, ")") -> 30
(29, ")") -> 30
(30, "do") -> 31
(31, comando) -> 31
(31, "end") -> 32
(32, "while") -> 12
```

../1–linguagem/notacoes/JFLAP/comando/comando.txt

E, finalmente, a saída referente à submáquina expressão:

```
initial: 0
final: 3, 4, 9
(0, "!") -> 1
(0, "-") -> 1
(0, "(") -> 2
(0, inteiro) -> 3
(0, float) -> 3
(0, "true") -> 3
(0, "false") -> 3
(0, char) -> 3
(0, identificador) -> 4
(1, "(") -> 2
(1, inteiro) -> 3
(1, float) -> 3
(1, "true") -> 3
(1, "false") -> 3
(1, char) -> 3
(1, identificador) -> 4
(2, expressao) -> 8
(3, "-") -> 0
(3, "^") -> 0
(3, "*") -> 0
(3, "/") -> 0
(3, "+") -> 0
(3, "==") -> 0
(3, "!=") -> 0
(3, "<") -> 0
(3, ">") -> 0
(3, "<=") -> 0
(3, ">=") -> 0
(3, "and") -> 0
(3, "or") -> 0
```

```
(4, "-") -> 0
(4, "(") -> 5
(4, "[") -> 6
(4, ".") -> 7
(4, "^") -> 0
(4, "*") -> 0
(4, "/") -> 0
(4, "+") -> 0
(4, "==") -> 0
(4, "!=") -> 0
(4, "<") -> 0
(4, ">") -> 0
(4, "<=") -> 0
(4, ">=") -> 0
(4, "and") -> 0
(4, "or") -> 0
(5, expressao) -> 11
(5, ")") -> 3
(6, expressao) -> 10
(7, identificador) -> 9
(8, ")") -> 3
(9, "-") -> 0
(9, "[") -> 6
(9, ".") -> 7
(9, "^") -> 0
(9, "*") -> 0
(9, "/") -> 0
(9, "+") -> 0
(9, "==") -> 0
(9, "!=") -> 0
(9, "<") -> 0
(9, ">") -> 0
(9, "<=") -> 0
(9, ">=") -> 0
(9, "and") -> 0
(9, "or") -> 0
(10, "]") -> 9
(11, ")") -> 3
(11, ",") -> 12
(12, expressao) -> 11
```

../1–linguagem/notacoes/JFLAP/expressao/expressao.txt

2.2.2 Lista dos autômatos

Nesta seção listaremos simplesmente os autômatos resultantes representados de uma maneira gráfica a partir das tabelas de transições da seção anterior.

A ferramenta utilizada para gerar a modelagem gráfica é o JFLAP, obtido do site (http://www.cs.duke.edu/csed/jflap/)

Figura 1 – Autômato representando a submáquina programa

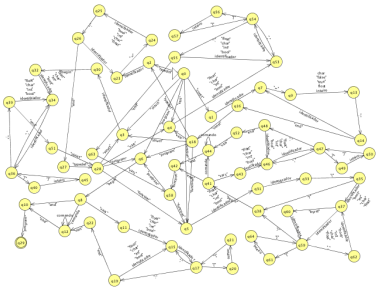


Figura 2 – Autômato representando a submáquina comando

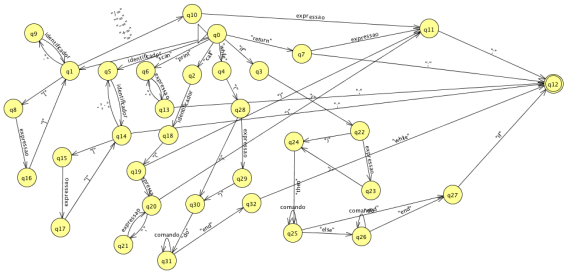
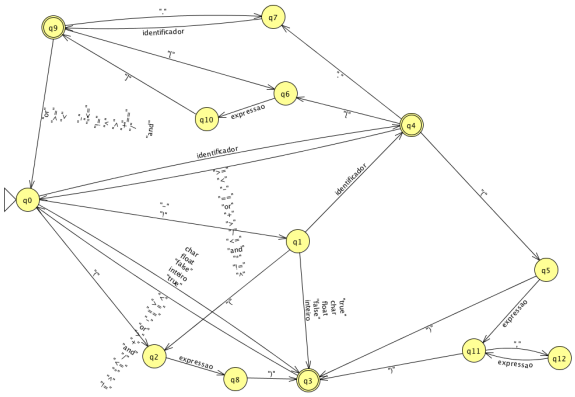


Figura 3 – Autômato representando a submáquina expressao



3 Comentários sobre a implementação

3.1 Desenvolvimento do módulo sintático

Partindo do modelo da notação de Wirth reduzida, nosso analisador sintático baseia-se na interação entre três máquinas de estados, representando as regras finais, que podem se chamar mutuamente, o que é previsto por estarmos trabalhando com um Autômato de Pilha Estruturado (APE).

O módulo sintático, portanto, começa suas atividades com a premissa que as três máquinas de estados já estejam criadas e inicializadas. Pressupõe-se que essa inicialização seja executada no programa principal (função main) do compilador.Essa inicialização ocorre de forma análoga ao que realizamos no módulo léxico e é baseada em tabelas de transições, as quais encontram-se descritas em arquivos externos, facilitando futuras modificações.

Utiliza-se a subrotina verify_syntax(), a qual já capaz de dizer se o compilador reconhece a linguagem. Mais detalhadamente, essa verificação tem como principal atividade o lançamento da função recognize(), utilizando a máquina que representa "programa‘’ e o primeiro token lido pelo analisador léxico como parâmetros.

Grosso modo, o que esta função realiza é, dado o token passado via parâmetro, consulta-se sua tabela de transições internas e verifica-se se há alguma transição possível. Se sim, a transição é realizada, o estado do autômato é atualizado e um novo token é lido.

Cuida lembrar que há casos em que a transição chama uma nova submáquina, o que também é especificado pela tabela. Quando isso ocorre, a nova máquina é chamada através da mesma função regconize, mas passando a nova máquina como parâmetro e o mesmo token por referência. Em outros termos, não há necessidade de lookahead e, por serem armazenados em memória quando da chamada de submáquina, os tokens são consumidos sob demanda.

Por outro lado, se não houver transição possível na tabela para um dado token, a função verifica se a máquina encontra-se em estado de aceitação, retornando verdadeiro, ou não, retornando falso. O valor falso traduz o caso em que houve um erro sintático na estrutura do código-

fonte e o erro é rapidamento propagado para as máquinas de estados que a chamaram, parando a execução do módulo. Já o valor verdadeiro, faz com que a máquina que a chamou continue sua execução a partir do próximo estado previsto pela tabela em que o autômato estava a esperar (resincronização); se o autômato é a raiz ("programa‘’) e já não há mais tokens a serem lidos, o compilador reconhece a sintaxe da linguagem.

3.2 Integração com o compilador

Devido à alta modularização configurada pela arquitetura escolhida, o programa principal que representa o compilador possui uma estrutura bastante simples.

Primeiramente, o módulo léxico é inicializado, passando o código-fonte a ser analidado como parâmetro. Em seguida, inicializa-se a Tabela de Símbolos e a Tabela de Palavras-chaves. Até aqui, não há diferenças em relação à entrega do analisador léxico.

A próxima ação, então, inicializa o módulo sintático, o que consiste basicamente na criação das máquinas de estados que representam as expressões de Wirth a partir dos arquivos externos, como descrito na seção anterior.

Finalmente, chama-se a subrotina verify_syntax(), a qual foi explicada anteriormente e é capaz de decidir se o código-fonte pertence ou não à lingugagem definida pela gramática.