

Tradutor - Trabalho Prático

João Lucas Azevedo Yamin Rodrigues da Cunha - 17/0013731

Universidade de Brasília
Brasília - DF, CEP 70910-900
jlyaminc@gmail.com

1 Motivação

A disciplina de Tradutores é uma das últimas da cadeia obrigatória do curso de Ciência da Computação. Ela aborda, em sua ementa, conteúdos de diversos períodos do curso. Por isso, a implementação do analisador léxico, bem como os demais componentes do projeto da disciplina a serem implementados futuramente, permite a assimilação da união do aprendizado acumulado ao longo do curso.

Para este projeto será utilizada a linguagem C-IPL, um subconjunto da linguagem C com adição de tratamento de listas. Dentro da computação, a lista é uma das estruturas de dados mais utilizadas e a adição dessa primitiva à linguagem já as disponibiliza de forma simples ao usuário, facilitando a programação. Elas possuem diversas aplicações práticas dentro e fora da matemática, além de servir como base para a implementação de outros conceitos.

2 Descrição da Análise Léxica

Para montagem do analisador léxico foi utilizada a ferramenta de geração de analisador léxico Flex [PWJ, Nac]. Redigiu-se um arquivo com extensão .l que serve de entrada para a ferramenta, descrevendo como deve ser feita a geração do analisador. Nele foram adicionadas diversas capturas de caracteres por meio de expressões regulares.

Os lexemas reconhecidos foram enviados como *tokens* para a etapa de análise sintática, com atributos adicionais de linha e coluna. As sequências de caracteres não identificadas são impressas na tela como erros léxicos.

Também no escaneamento léxico, foi realizada uma implementação parcial de escopo: a cada { encontrado, é incrementado um contador. Este serve como identificador para referenciar o escopo dos símbolos na tabela de símbolos. Para determinar a hierarquia dos escopos, foi definida uma lista encadeada, em que cada novo escopo possui um ponteiro para seu pai (o escopo acima). Assim, é possível navegar aos escopos anteriores para definir onde foram definidas as variáveis. É feita a impressão da hierarquia dos escopos após a impressão da tabela de símbolos.

No anexo, a tabela da Seção B apresenta a descrição dos *tokens* utilizados.

3 Descrição da Análise Sintática

A partir da gramática presente na Seção A do anexo e por meio da ferramenta Bison [Fou], implementou-se um analisador sintático. Integrado com o analisador léxico, o sintático identifica quaisquer erros sintáticos presentes no código analisado e, com o auxílio de funções adicionais implementadas, permite a criação de uma árvore de derivação simplificada e de uma tabela de símbolos simples.

A árvore foi implementada utilizando *structs* compostas por um *token* (caso seja um nó-folha), um nome e um vetor de filhos (ambos caso seja um nó-pai) para representar os elementos. Além disso, para uso na Análise Semântica, os nós possuem um campo que identifique o seu tipo e um campo que indica a necessidade de conversão de tipo. Os nós são adicionados quando se identificam regras da gramática que possuem terminais (nós-folha) ou ramificações (nós-pai). Não são adicionados todos os não-terminais, apenas os que foram julgados importantes para a interpretação de sua estrutura. Sua impressão é feita de forma recursiva.

No caso da tabela de símbolos, a inserção é feita sempre que uma declaração de variável ou de função é encontrada. Uma entrada com identificador, tipo, número identificador do escopo e um indicador da natureza da declaração (variável, função ou parâmetro) é inserida nesses casos. A tabela de símbolos foi construída utilizando uma lista encadeada.

Apesar da incrementação de escopo ser realizada na etapa léxica, o retorno ao escopo anterior é realizado pelo analisador léxico. Foi necessário utilizar uma ação no meio de uma regra sintática (*mid-rule action*) para permitir que a troca de escopo fosse realizada somente mediante o processamento de todos os elementos anteriores ao delimitador de escopo (`}`).

Ao fim da análise sintática são impressos: os erros encontrados, a árvore de derivação simplificada (caso não hajam erros sintáticos), a tabela de símbolos e a hierarquia de escopo, nesta ordem.

4 Descrição da Análise Semântica

A Análise Semântica é realizada em apenas uma passagem, em conjunto com a Análise Sintática e Léxica. A anotação de tipo é realizada a partir dos nós terminais, onde um atributo sintético é gerado de acordo com o tipo destes terminais. A partir disso, é decidida a tipagem das expressões, anotando também as conversões de tipo necessárias. Essas conversões serão feitas nos casos em que seja possível converter um tipo *int* para *float* ou o contrário. Nesses casos, é adicionada uma notação no nó que indique essa ação. Para cada identificador, o tipo e o escopo respectivos estão presentes na árvore e na tabela de símbolos.

As verificações de escopo (que seguem a hierarquia comentada na Seção 3) permitem identificar se houve a declaração inicial e/ou redeclarações de identificadores e se a função `main` está presente. Todas essas funcionalidades se dão através da varredura da tabela de símbolos.

Quanto à passagem de parâmetros, é feita uma busca na tabela de símbolos para recuperar os parâmetros da função chamada. Em seguida, é realizada uma

comparação um-a-um entre os parâmetros e os argumentos passados. Nesse momento são adicionadas conversões implícitas de tipo se necessário.

5 Descrição da Geração de Código Intermediário

A Geração de Código Intermediário é realizada somente se não houver erros nas etapas de análise. Caso o código de entrada esteja correto, é criado um arquivo de mesmo nome com extensão `.tac`. Nele, são impressas duas seções: uma seção `.table` e uma seção `.code`.

Na seção `.table`, são adicionadas as declarações de variáveis a partir da tabela de símbolos. E, junto do identificador da variável é concatenado o número do escopo correspondente com o objetivo de evitar redeclaração de variáveis. As variáveis de tipo simples são declaradas normalmente. Já as variáveis com tipo de lista serão declaradas com o tipo de seus elementos (*int* ou *float*) e inicializadas com zero. O intuito é fazer dessas declarações um ponteiro e usar da alocação dinâmica para atualizar os elementos da lista.

Para a seção `.code`, é analisada a árvore gerada pelas etapas anteriores. A partir de cada nó da árvore, serão adicionados trechos de código correspondente à instrução analisada. A maior parte das instruções serão traduzidas de forma direta, utilizando as variáveis temporárias onde necessário. Para as declarações de funções serão utilizados rótulos de mesmo nome da função e, para seus parâmetros, será utilizada a instrução `param`. Os rótulos também serão utilizados para laços de repetição: tanto para o `for`, o `write` e o `writeln` (para imprimir mais de um caractere), quanto para as operações `filter` e `map`.

Como dito acima, o comportamento das listas é tratado a partir da alocação dinâmica. Para adição de um novo elemento com o construtor, é realizada uma nova alocação adicionando um espaço e o atualizando o ponteiro. Para a remoção de um elemento com o destrutor, o comportamento é semelhante, mas alocando um espaço a menos. As demais operações serão realizadas ao referenciar os espaços do vetor alocado.

A utilização de variáveis temporárias se dá de forma rotativa: inicia-se um contador em `$0` e incrementa até `$9` antes de reiniciar a sequência. Para montagem de expressões, o contador é incrementado para o endereço de destino da instrução e é decrementado para o endereço do operando. Dessa forma, garante-se que o valor da expressão será mantido durante as sequências de instruções das expressões.

6 Arquivos de Teste

Dentro do diretório `tests/`, foram criados dois arquivos de teste corretos:

1. `correct_test_01.c`;
2. `correct_test_02.c`, .

E dois arquivos de teste contendo erros:

1. `incorrect_test_01.c`:
 - Erro semântico: quatro argumentos passados para função de três parâmetros na linha 6, coluna 32;
 - Erro semântico: dois argumentos passados para função de três parâmetros na linha 7, coluna 24;
 - Erro semântico: argumento de tipo `int` incompatível com parâmetro de tipo `int list` na linha 8, coluna 30; e
 - Erro sintático: declaração de variável sem ponto e vírgula na linha 12, coluna 1.
2. `incorrect_test_02.c`:
 - Erro sintático: Declaração de variável dentro da expressão condicional do `if` na linha 3, coluna 10;
 - Erro sintático: Expressão de soma dentro de `read` na linha 6, coluna 14;
 - Erro semântico: Uso do operador `?` em variável inteira na linha 7, coluna 18;
 - Erro semântico: Expressão de tipo inválida para a função `writeln` na linha 8, coluna 6; e
 - Erro semântico: Retorno de tipo `float list` em função de tipo `int` na linha 11, coluna 2.

7 Compilação e Organização

O código referente a este projeto foi desenvolvido e executado em um sistema com as seguintes características:

- Sistema Operacional: Manjaro 21.10;
- Kernel: 5.10 LTS;
- gcc: 11.1.0;
- ld: 2.36.1;
- flex: 2.6.4;
- make: 4.3.

Para execução do tradutor, o comando

```
$ make compile
```

compilará o código e gerará um executável de nome ‘tradutor’. Este comando é equivalente a executar

```
$ bison -d ./src/cipl_syn.y
$ flex ./src/cipl_lex.l
$ gcc -c ./src/symbol.c
$ gcc -c ./src/tree.c
$ gcc -c ./src/scope.c
$ gcc -c ./src/type.c
$ gcc -c ./src/gen.c
$ gcc -o tradutor cipl_syn.tab.c lex.yy.c symbol.o scope.o
tree.o type.o gen.o -g -Wall -Wextra -Wpedantic
```

Com isto, execute utilizando

```
$ ./tradutor <caminho_para_o_arquivo>
```

Como dito acima, os testes se encontram no diretório `./tests/`. No mesmo diretório em que se encontra o arquivo de entrada, será adicionado o arquivo gerado com a extensão `.tac`.

Referências

- [ALSU06] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2006.
- [Fou] Free Software Foundation. GNU Bison - The Yacc-compatible Parser Generator. <https://www.gnu.org/software/bison/manual/>. Acessado por último em 1 Set 2021.
- [Jon] D. Jones. The New C Standard. http://www.coding-guidelines.com/cbook/cbook1_1.pdf. Acessado por último em 18 Ago 2021.
- [Nac] V. Nachiappan. USING LEX. <https://silcnitc.github.io/lex.html>. Acessado por último em 10 Ago 2021.
- [Pol] B. Pollack. BNF Grammar for C-Minus. <http://www.csci-snc.com/ExamplesX/C-Syntax.pdf>. Acessado por último em 10 Ago 2021.
- [PWJ] V. Paxson, Estes W., and Millaway J. Flex Manual. <https://www.cs.virginia.edu/~cr4bd/flex-manual/index.html>. Acessado por último em 7 Out 2021.
- [SK] K. Slonneger and B Kurtz. Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach. <https://homepage.divms.uiowa.edu/~slonnegr/plf/Book/Chapter1.pdf>. Acessado por último em 2 Set 2021.

A Gramática

A Gramática a seguir foi gerada utilizando como base a gramática C-Minus [Pol]. Os rótulos em letras todas maiúsculas são equivalentes aos apresentados na tabela B.

1. $program \rightarrow declarationList$
2. $declarationList \rightarrow declarationList\ declaration \mid declaration$
3. $declaration \rightarrow variableDeclaration \mid functionDeclaration$
4. $variableDeclaration \rightarrow \mathbf{TYPE\ ID\ ;}$
5. $functionDeclaration \rightarrow \mathbf{TYPE\ ID\ (params)}$
 $\quad compoundStmt$
6. $params \rightarrow paramList \mid \epsilon$
7. $paramList \rightarrow paramList, param \mid param$
8. $param \rightarrow \mathbf{TYPE\ ID}$
9. $compoundStmt \rightarrow \{ statementList \}$

10. $statementList \rightarrow statementList \text{ statement } | \epsilon$
11. $statement \rightarrow expressionStmt | compoundStmt | conditionalStmt | loopStmt | returnStmt | variableDeclaration | inOutStmt$
12. $expressionStmt \rightarrow expression ; | ;$
13. $conditionalStmt \rightarrow \text{if} (expression) conditionalBody | \text{if} (expression) conditionalBody \text{ else } conditionalBody$
14. $conditionalBody \rightarrow expressionStmt | compoundStmt | conditionalStmt | loopStmt | returnStmt | inOutStmt$
15. $loopStmt \rightarrow \text{for} (expression ; logicExpression ; expression) statement$
16. $returnStmt \rightarrow \text{return } expression ;$
17. $inOutStmt \rightarrow \text{INPUT} (ID); | \text{OUTPUT} (outputArgs);$
18. $expression \rightarrow ID = expression | logicExpression$
19. $logicExpression \rightarrow logicExpression \text{ OP_LOGIC } relatExpression | relatExpression$
20. $relatExpression \rightarrow relatExpression \text{ OP_RELAT } listExpression | listExpression$
21. $listExpression \rightarrow addExpression \text{ OP_LIST } listExpression | addExpression$
22. $addExpression \rightarrow addExpression \text{ OP_ADD } mulExpression | mulExpression$
23. $mulExpression \rightarrow mulExpression \text{ OP_MUL } factor | factor$
24. $factor \rightarrow (expression) | unaryExpression | call | ID | \text{FLOAT} | \text{INT} | \text{NIL}$
25. $unaryExpression \rightarrow \text{UN_OP } factor | \text{OP_ADD } factor$
26. $call \rightarrow ID (args)$
27. $outputArgs \rightarrow \text{STRING} | expression$
28. $args \rightarrow argList | \epsilon$
29. $argList \rightarrow argList , expression | expression$

B Tokens e Lexemas

Rótulo do Token	Padrão do Lexema (RegEx)	Lexema de Exemplo
ID	<code>[_a-zA-Z][_a-zA-Z0-9]*</code>	num
DIGIT	<code>[0-9]</code>	88
FLOAT	<code>(-)?{DIGIT}*.{DIGIT}+</code>	-402.3
INT	<code>(-)?{DIGIT}+</code>	25
OP_ADD	<code>[+-]</code>	+
OP_MUL	<code>[*/]</code>	/
OP_LOGIC	<code>(&&) ()</code>	&&
OP_RELAT	<code>(<) (<=) (>) (>=) (<=) (<=)</code>	>=
OP_ASSIG	<code>(=)</code>	=
UN_OP	<code>!(%)(?)</code>	!
OP_LIST	<code>(>>) (<<) (:)</code>	>>
TYPE	<code>(int) (float) (int list) (float list)</code>	int list
NIL	<code>(NIL)</code>	NIL
IF	<code>(if)</code>	if
ELSE	<code>(else)</code>	else
FOR	<code>(for)</code>	for
RETURN	<code>(return)</code>	return
INPUT	<code>(read)</code>	read
OUTPUT	<code>(write) (writeln)</code>	writeln
SEMICOLON	<code>(;)</code>	;
COMMA	<code>(,)</code>	,
CURLYB	<code>[{ }]</code>	{
PARENTHESIS	<code>[()]</code>	(
STRING	<code>(" .*") (' . * ')</code>	"string"

Tabela 1. Tokens e lexemas de exemplo