Analisador Semântico

wladus-lang

Wladimir Gramacho - 15/0048718

Tradutores - Profª. Cláudia Nalon

Novembro de 2019

1 Contexto

O curso de Tradutores ministrado na Universidade de Brasília tem como objetivo o estudo dos componentes de um tradutor assim como a implementação destes[1]. Sendo assim, o trabalho desta matéria consiste na construção de um tradutor para uma linguagem de programação simplificada e de propósito específico. A linguagem que está sendo criada se chama `wladus-lang` e tem como objetivo possibilitar a interpolação de strings e criação de strings como tipo nativo da linguagem. O desenvolvimento do projeto se dá em 6 etapas diferentes, sendo elas: Escolha do Tema, Analisador Léxico, Analisador Sintático, Analisador Semântico, Gerador de Código Intermediário e Apresentação do Trabalho. Este relatório se trata da apresentação da quarta fase: Analisador Semântico.

O analisador semântico tem como objetivo a verificação da corretude semântica no código. Algumas verificações são:

- Checagem de tipos: avaliar se atribuições e operações tem tipos equivalentes.
- Declaração: checar se a variável ou função foi declarada antes de ser usada e se foi declarada mais de uma vez no mesmo escopo.
- Parâmetros de funções: passagem correta de parâmetros, em relação ao tipo e aridade.
- Operações inválidas: divisão por zero, por exemplo.

2 Refatorações

Antes de começar o desenvolvimento das verificações semânticas, foi necessário refatorar o analisador sintático de forma a melhorar o desenvolvimento das etapas subsequentes. A principal mudança ocorreu na construção da árvore sintática. A árvore estava

[1] `https://matriculaweb.unb.br/graduacao/disciplina.aspx?cod=116459`

sendo construída com diferentes tipos de nós para cada estrutura da linguagem. Após a revisão da professora, optou-se pela utilização de somente um tipo de nó na árvore, que contém o tipo da estrutura do nó, o tipo (inteiro, decimal, *string* ou vazio) do nó, seus filhos esquerdo e direito e um campo opcional com alguma informação adicional do nó. Dessa maneira, a construção da árvore ficou muito mais fácil.

Outra modificação necessária foi na tabela de símbolos. As entradas na tabela de símbolos tinham que suportar escopo. Dessa forma, cada função declarada no programa abre um novo escopo e variáveis da função (parâmetros e variáveis locais) são adicionadas à tabela de símbolos da função. Variáveis globais do programa permanecem na tabela de símbolos global. O escopo foi implementado através de uma pilha de escopo, que tem um escopo empilhado toda vez que inicia-se a declaração de uma função e desempilhado ao terminar de ler a declaração da função.

3 Desenvolvimento

Uma vez implementado o escopo, foi feita a checagem da declaração de variáveis e funções. Essa checagem avalia se a variável ou função chamada já foi declarada e se ela já não foi declarada anteriormente. Essa verificação foi feita ao percorrer a tabela de símbolos e o escopo no qual a variável estava contida. Se for encontrado erro, o compilador retorna uma mensagem de erro explicativa do erro, localizando a linha da ocorrência.

Outra checagem implementada foi a verificação de tipos. Uma vez que um nó da árvore sintática possui um campo para o tipo do nó, foram verificados incompatibilidades em atribuições a variáveis e operações binárias. Por ser uma linguagem fortemente tipada e sem conversão de tipos, `wladus-lang` não permite que se façam operações entre tipos de dados diferentes. Logo, a operação `3 + 0.14` causaria um erro, uma vez que `3` é um número inteiro e `0.14` um número decimal. Uma variável inteira, por exemplo, somente poderá receber valores inteiros.

No momento de entrega do projeto prático, o autor da linguagem não havia lembrado das outras verificações semânticas necessárias, apresentadas na Seção 1. Portanto, as outras verificações semânticas serão apresentadas na última etapa do projeto.

3.1 Tabela de Símbolos

A tabela de símbolos foi construída no formato de uma *hash table*. Para isso, foi utilizada a biblioteca uthash[2] para o auxílio na implementação. O escopo foi implementado com o auxílio da biblioteca utstack[3] Com as modificações feitas, são armazenadas as seguintes informações na tabela de símbolos:

- Nome do identificador. Esse campo é utilizado como chave da *hash table*, de modo que seja possível procurar diretamente pelo nome do identificador na tabela de

[2] `https://troydhanson.github.io/uthash/`
[3] `https://troydhanson.github.io/uthash/utstack.html`

símbolos, obtendo uma complexidade de tempo constante para busca e inserção na tabela.

- Caractere que representa o tipo do identificador (‘i’nt, ‘f’loat, ‘v’oid ou ‘s’tring).
- Caractere que representa o tipo do objeto (‘V’ariável ou ‘F’unção).
- Campos para funções:

  - Ponteiro para nó na árvore sintática. Esse campo é usado para que seja possível acessar o corpo de uma função declarada.

  - Ponteiro para uma lista de entradas na tabela de símbolos do escopo da função. Esses nós são mais simples, contendo somente o nome do identificador, seu tipo, seu tipo de objeto (‘V’ariável ou ‘P’arâmetro).

- *Handle* necessário para a criação da *hash table* segundo a implementação do uthash.

A inserção na tabela de símbolos foi modificada para que considerasse escopo e gerasse erros caso houvessem. Além disso, foi corrigido um erro da etapa anterior no processo de liberação de memória da tabela de símbolos.

3.2 Árvore Sintática

Além das modificações apresentadas na Seção 2, foram necessárias mudanças ao adicionar nós à árvore. Um exemplo é a inserção do nó ao ler um nó condicional. Como é um nó que tem três filhos, a estrutura do nó ficou:

- Filho esquerdo: aponta para um novo nó, que possui a expressão da condição e o corpo do `if`.
- Filho direito: se a condição tiver um `else`, aponta para o corpo do `else`. Se não, aponta para nulo.

Com essa lógica, foi possível fazer com que todos os nós tivessem a mesma estrutura e ainda estivessem conectados. As implementações de imprimir e liberar a memória da árvore também foram simplificadas, uma vez que não era necessário tanta especificade pelos nós terem a mesma estrutura.

3.3 Gramática

Foram necessárias algumas mudanças na gramática. Vale ressaltar que essa gramática é inspirada na gramática do padrão C ANSI apresentada por Jutta Degener em 1995, disponível no site da Linkoping University[4]. As mudanças feitas foram:

- Remoção das regras *param-list* e *param*. Agora *params* produz diretamente os identificadores com seus tipos.

[4] `https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`

- *local-declarations* não produz mais *var-declaration* e sim *local-var-declaration*, que tem uma forma diferente de construir o nó na árvore sintática.
- Adição do token `error` à produção da regra *expression-stmt*, possibilitando a recuperação de erros.
- *term* não produz mais *expression* e sim *simple-expression*, uma vez que a construção `a = b = 3` não é válida em `wladus-lang`.
- Identificadores podem ter *underscore*.

Portanto, a gramática atual tem a forma:

1. *program* → *declarations*
2. *declarations* → *declarations declaration* | *declaration*
3. *declaration* → *var-declaration* | *func-declaration*
4. *var-declaration* → **TYPE ID ;**
5. *fun-declaration* → **TYPE ID (** *params* **)** *compound-stmt*
6. *params* → *params* **,** **TYPE ID** | **TYPE ID** | ε
7. *compound-stmt* → **{** *local-declarations stmt-list* **}**
8. *local-declarations* → *local-declarations local-var-declaration* | ε
9. *local-var-declaration* → **TYPE ID ;**
10. *stmt-list* → *stmt-list stmt* | ε
11. *stmt* → *expression-stmt* | *conditional-stmt* | *iteration-stmt* | *return-stmt*
12. *expression-stmt* → *expression* **;** | *error* **;**
13. *conditional-stmt* → **if (** *expression* **)** *compound-stmt* |
    if (*expression* **)** *compound-stmt* **else** *compound-stmt*
14. *iteration-stmt* → **while (** *expression* **)** *compound-stmt*
15. *return-stmt* → **return** *expression* **;** | **return ;**
16. *expression* → *var* = *expression* | *simple-expression*
17. *var* → **ID**
18. *simple-expression* → *op-expression relop op-expression* | *op-expression*
19. *relop* → <= | < | > | >= | == | **!=**

20. *op-expression* → *op-expression op term* | *term*
21. *op* → **+** | **-** | **/** | *****
22. *term* → **(** *simple-expression* **)** | *var* | *call* | **“** *string* **”** | **NUM** | **DEC**
23. *call* → **ID** **(** *args* **)** | **write** **(** *simple-expression* **)** | **read** **(** *var* **)**
24. *string* → *string* **STRING** | *string* **#{** *expression* **}** | ε
25. *args* → *arg-list* | ε
26. *arg-list* → *arg-list* **,** *simple-expression* | *simple-expression*

ID = $letter\ (letter|digit|underscore)^*$

NUM = $digit^+$

DEC = $(digit)^+.(digit)^+$

TYPE = $(int|float|void|string)$

STRING = $(\backslash\backslash.|[^{\wedge}\#\{\}\backslash\backslash])^*$

$letter = a \mid ... \mid z \mid A \mid ... \mid Z$

$digit = 0 \mid ... \mid 9$

$whitespace = \backslash n \mid \ \mid \backslash t$

$comment = "//".^*$

$underscore = \_$

3.4 Tratamento de Erros

Ao identificar os erros apresentados na Seção 3, o programa mostra uma mensagem de erro e se recupera do erro, conseguindo prosseguir a leitura do código fonte e identificar mais erros no código. Isso foi implementado ao colocar o token `error` como uma das produções da regra *expression-stmt*. O `error` é um token especial do Bison que sempre é definido e reservado para tratamento de erros [1]. O analisador Bison gera um token de erro sempre que ocorre um erro de sintaxe.

4 Arquivos de teste

Junto com os analisadores léxico, sintático e semântico e o arquivo de diretivas para execução (*makefile*), os arquivos compactados enviados na submissão dessa etapa contêm arquivos de teste do analisador. Há três arquivos de teste com entradas válidas (`valid0.w`, `valid1.w` e `valid2.w`). Esses arquivos testam todas as estruturas corretas da gramática.

Além disso, há seis arquivos inválidos (`invalid1.w`, `invalid2.w`, `invalid3.w`, `invalid4.w`, `invalid5.w` e `invalid6.w`), com diferentes tipos de erros. Seguem os erros:

1. `invalid1.w`: erros sintáticos na linha 4 e linha 5.
2. `invalid2.w`: error sintático na linha 2.
3. `invalid3.w`: erros semânticos, linha 3 com variável já declarada, linha 5 com variável não declarada e linha 7 com função não declarada.
4. `invalid4.w`: erros semânticos, linhas 2, 5 e 9 com variáveis já declaradas e linha 8 com função redeclarada.
5. `invalid5.w`: erros semânticos, linhas 10 até 14 com erros de incompatibilidade de tipos e linha 16 com função não declarada.
6. `invalid6.w`: erro semântico, linha 8 com incompatibilidade de tipos.

Para compilação dos analisadores e execução dos testes, entre na pasta `src` e execute no terminal de comando as seguintes linhas:

```
make all

make test
```

A saída da execução dos testes será os erros do programa, se encontrados. Se, ao executar, for passada a opção `-tt`, a árvore de símbolos e a árvore sintática anotada serão impressas. Os exemplos no makefile têm essa opção definida por padrão. Se for encontrado qualquer erro no programa, a árvore de símbolos e a árvore sintática não serão impressas, mesmo que a opção estiver habilitada.

4.1 Dificuldades

Além das dificuldades inerentes ao desenvolvimento de uma linguagem, a principal dificuldade desta etapa foi a refatoração da árvore sintática, uma vez muito código foi escrito para essa funcionalidade na etapa anterior e foi complicado entender como seria a construção de uma árvore usando somente uma estrutura de nó.

Passada essa dificuldade, foi necessária muita atenção na criação de escopos na tabela de símbolos. Essa funcionalidade tomou bastante tempo, mas uma vez implementada corretamente, facilitou muito a verificação de erros.

Referências

[1] Charles Donnelly and Richard Stallman. Gnu bison manual (version 3.4). `https://www.gnu.org/software/bison/manual/bison.pdf`, 2019.