Gerador de Código Intermediário

Pedro Henrique Santos Gonzaga - 140030131

Departamento de Ciência da Computção, Universidade de Brasília (UnB) - Maio/2021

1 Motivação e Proposta

Este relatório tem como objetivo apresentar a última parte do projeto da disciplina de Tradutores do curso de Ciência da Computação da Universidade de Brasília. O projeto é dividido em quatro partes e tem como finalidade a construção das etapas de análise de um compilador e seu gerador de código de três endereços. Nesta quarta e última etapa será construído o gerador de código de três endereços. A linguagem foco do projeto é a linguagem C com a adição de duas novas primitiva, que tornam possíveis determinadas operações entre conjuntos. Com estas novas primitiva, o leque de possibilidades para a linguagem se expande, pois, em C puro, para se trabalhar com conjuntos, é necessária a declaração e manipulação de várias estruturas de dados, não havendo um padrão para tal. Portanto, a definição formal destas novas primitivas permite que suas operações sejam padronizadas e facilitadas.

2 Descrição da Análise Léxica

Para o desenvolvimento do analisador léxico, foram utilizadas expressões regulares para identificar os lexemas aceitos pela linguagem, assim como seus tokens auxiliares (pontuação, chaves, etc). As definições de todas as regras podem ser vistas no arquivo *skylang.l*. Quando o analisador é executado e lhe é dado um arquivo de entrada para análise, o mesmo inicia a leitura dos caracteres do arquivo seguindo as regras definidas pelas expressões regulares. Caso um lexema que não esteja previsto dentro das regras estipuladas seja encontrado, a função *LexicalError()* é chamada e uma mensagem de erro é mostrada ao usuário, explicitando a linha e a coluna do lexema inválido. O papel do analisador léxico é tão somente verificar os lexemas do arquivo de entrada, nenhuma estrutura de dados adicional foi necessária nesta etapa.

3 Descrição da Análise Sintática

Para o desenvolvimento da análise sintática, a gramática definida para o projeto foi escrita em forma de código utilizando-se a ferramenta **BISON** [4] no arquivo skylang.y. Foi feita a integração com o analisador léxico (skylang.l) de forma que os tokens analisados são passados para o analisador sintático, que por sua vez avalia a sequência de tokens de acordo com a gramática e gera uma árvore

sintática abstrata. A árvore foi construída de forma que cada nó tem, no máximo, 5 filhos. Cada nó contém informações sobre o tipo do nó, o nome da derivação daquele nó e alguns outros campos sinalizadores que serão explicados mais a fundo na seção de análise semântica. Caso um erro sintático seja encontrado, a função *yyerror()* é chamada, apontando a linha e a coluna do erro sintático. Para que a integração do analisador léxico e sintático fosse possível, foi criada uma union no skylang.y, que permitiu passar as informações dos tokens entre os dois analisadores. A gramática definida na etapa anterior também sofreu algumas mudanças, já que alguns erros ainda estavam com a correção pendendte.

4 Descrição da Análise Semântica

Para realizar a análise semântica, foi utilizado o algoritmo de duas passagens. Foi criada uma variável global "passagem"para controlar qual parte do código deve ser executado em cada etapa. Na primeira passagem pelo código, é gerada a tabela de símbolos contendo os nomes, tipos e escopo de cada símbolo. Na segunda passagem, é feita a análise de fato. Para checar o escopo de cada símbolo, foi definida uma variável "EscopoAtual"que contém o escopo atual do código. Ao se encontrar um símbolo, é checado se o símbolo existe na tabela de símbolos. Caso exista, a variável "EscopoAtual"é comparada com o escopo da variável. Caso o escopo esteja errado, um erro é mostrado. O mesmo acontece caso o símbolo não exista na tabela de símbolos, ou seja definido duas vezes no código (no mesmo escopo). Os erros semânticos de conversão de tipo proibidas, operações proibidas para determinado tipo e a conversão de tipo implícita são todas feitas por meio do percorrimento da árvore sintática abstrata anotada. Para que isso fosse possível, a árvore sintática abstrata definida na seção de análise sintática sofreu algumas mudanças. Foram adicionados os campos "conversão"(para sinalizar quando houver uma conversão e para checar se ela é permitida) e "value"para armazenar o valor do ID, da constante, ou do operador daquele nó.

5 Política de conversão de tipos

A conversão de tipo implícita foi definidada da seguinte forma: É possível converter variáveis e constantes do tipo **float** para **int** e vice-versa. Variáveis do tipo **elem** são polimórficas e podem ser convertidas tanto para **float** quanto para **int** e **set** . Variáveis do tipo **set** não podem ser convertidas para nenhum outro tipo. Conversões possíveis:

- Atribuições: Em atribuições, o valor à direita da atribuição sempre será convertido para o valor à esquerda.
- Operações: Em operações aritméticas, as conversões sempre serão feitas convertendo os elementos mais à direita da expressão para o tipo dos elementos mais à esquerda.
- Conversões possíveis:
  1 = intToFLoat

2 = floatToInt
3 = elemToInt
4 = IntToElem
5 = elemToFloat
6 = floatToElem

6 Tabela de símbolos

Para a implementação da tabela de símbolos foi utilizada uma biblioteca C que permite a criação uma tabela *Hash* [4], facilitando o processo de busca e adição dos símbolos à tabela (a *Hash table* também é mais performática do que salvar os símbolos em um vetor, por exemplo). A estrutura dessa tabela pode ser vista no arquivo *symboltable.h*. A tabela de símbolos armazena todas as funções e variáveis declaradas pelo programa, seus nomes, tipo (int, float, elem, ou set) e tipo do símbolo (variável, ou função), assim como um identificador único para cada entrada da tabela. Para esta nova etapa do projeto, a tabela de símbolos foi modificada para que fosse possível guardar também o registrador a ser utilizado por cada variável no código intermediário.

7 Tratamento de Erros Semânticos

O analisador semântico implementado é capaz de apontar os seguintes erros:

1. Chamada de função não declarada;
2. Uso de variáveis não declaradas;
3. Uso de variáveis fora do seu escopo permitido;
4. Códigos sem a função main();
5. Funções com retorno do tipo errado;
6. Redeclaração de funções;
7. Redeclaração de variáveis;
8. Operações não permitidas para o tipo SET;
9. Chamada de funções com passagem de parâmetros do tipo errado;
10. Chamada de funções com o número errado de parâmetros passados.

8 Geração de Código Intermediário

Para a geração de código intermediário, é feita, durante a primeira passagem, a tradução dos comandos em C para TAC (Three Address Code). Para armazenar as linhas do código intermediário,à medida em que elas são geradas ,foi utilizada uma lista de strings, com o auxílio da biblioteca utlist.h, que possui macros para inserção, deleção e percorrimento de listas. No modelo proposto por este trabalho, o cabeçalho .table não conterá os rótulos das variáveis declaradas pelo código, já que a cada declaração de variável, um registrador será alocado para a

mesma. Para fazer este controle, foi criado um novo campo na tabela de símbolos (campo registrador), de forma que, à medida que variáveis são declaradas, uma variável global é incrementada, definindo assim o próximo registrador livre. Um novo campo na árvore abstrata também foi criado, o campo "resultado", que armazena e propaga para a próxima operação o registrador utilizado para salvar o valor da última operação. A nova primitiva *set* e suas operações serão tratadas da seguinte forma: Uma variável do tipo set será considerada um vetor de inteiros, ou float dentro do TAC, de forma que as operações de **add** serão feitas adicionando um elemento na próxima posição vazia do vetor. As operações de **remove** serão feitas percorrendo o vetor e removendo o elemento, caso ele esteja dentro dentro do conjunto. O operador **isSet** vai verficar se determinada variável é, ou não um vetor (ou seja, conjunto). Variáveis do tipo *elem* são automaticamente convertidas para *int* ou *float* no TAC: Caso o número tenha casas decimais, ele é convertido para *float*, caso não tenha, é convertido para *int* .

9 Arquivos de Teste

Juntamento com o código foram disponibilizados quatro arquivos de texto para teste. Os arquivos testeCorreto1.txt e testeCorreto2.txt possuem exemplos de código que não possuem erros sintáticos, léxicos, ou semânticos dentro da gramática definida para o projeto. Os arquivos testeErro1.txt e testeErro2.txt apresentam códigos com erros sintáticos, léxicos, ou semânticos. Os erros do arquivo testeErro1 são:

- Lexema $ não reconhecido na linha 3, coluna 5
- Erro Sintático na linha 3, coluna 6
- Erros semânticos na linha 30, colunas 10,14,20 e 22 (variável não declarada)
- Erro Sintático na linha 35, coluna 10 (ponto e vírgula inesperado)
- Erro Semântico: Tipo de retorno da função teste incorreto na linha 8
- Erro Semântico: Redeclaração da variável x na linha 24
- Erro Semântico: Número de argumentos incorretos para a chamada de função na linha 33, coluna 13.

Os erros do arquivo testeErro2 são:

- Erro semântico linha 2, coluna 7: variável utilizada no escopo errado
- Erro semântico linha 2, coluna 11: variável utilizada no escopo errado
- Erro sintático na linha 5, coluna 2: } inesperada
- Erro semântico linha 13, coluna 14: Variável não declarada
- Erro semântico: Tipo de retorno da função teste incorreto na linha 4;

10 Compilação e execução

O projeto foi originalmente desenvolvido em um SO Linux Ubuntu 20.04, utilizando a versão 3.7.2 do Bison e a versão 2.6.4 do Flex. Para a compilação manual

dos analisadores e do gerador de código intermediário, em ambiente Linux, abra o terminal, navegue até a pasta com os arquivos .l, .y e .txt e digite os seguintes comandos:

1 - flex skylang.l

2 - bison -d -v skylang.y

3 - gcc lex.yy.c skylang.tab.c -Wall -o skylang

Para executar o programa e realizar a análise dos arquivos de teste, digite o seguinte comando, também no terminal:

1- echo "arquivoDeTeste.txt"| ./skylang

Exemplo: echo testeCorreto1.txt | ./skylang

Alternativamente, pode-se utilizar o comando "make compile"para a compilação do código e os comandos make testeCorreto1, make testeCorreto2, make testeErrado1 e make testeErrado2 para realização dos testes. Depois de compilado e executado, o programa gerará um arquivo chamado skylang.tac, contendo o código intermediário proveniente da tradução do código C lido. Para testar o skylang.tac, digite no terminal, dentro do diretório contendo o arquivo .tac e o executável tac.so, o seguinte comando: ./tac.so skylang.tac

Referências

1. Alfred, Aho., Monica, Lam., Ravi, Sethi., Jeffrey, Ullman: Compiladores: Princípios, Técnicas e Ferramentas. 2nd edition. Pearson, 2008

2. Vern Paxson, Flex Manual, https://westes.github.io/flex/manual/. Último Acesso 20 Março 2021
3. Robert Corbett, Bison Manual, https://www.gnu.org/software/bison/manual/. Último acesso 5 Abril 2021
4. Hanson, Troy D.,Arthur O'Dwyer, Uthash Documentation, https://troydhanson.github.io/uthash/userguide.html. Último acesso 5 Abril 2021

A Gramática

A gramática da etapa anterios sofreu algumas mudanças, para que fosse possível corrigir os erros apontados pela professora.

1. *Programa* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *variable declaration* | *function declaration*
4. *variable declaration* → **Type ID**
5. *function declaration* → **Type ID** *(params) codeBlock*
6. *params* → params-list
7. *params-list* → *params-list* , *param* | *param*
8. *param* → **Type ID**
9. *codeBlock* → *StatementList*
10. *SatementList* → *StatementList Statement*
11. *statement* → *variable declaration* | *exp* | *ifStatement* | *ForallStatement* | *outPutStatement* | *inputStatement* | *callFuncStatement* | **Return** exp | forStatement;
12. *exp* → *SetExp* | *aritExp* | *relExp* | *assignmentExp* | *terminal* | *(exp)* | *!exp*
13. *assignmentExp* → *terminal* = *exp* | *terminal* = *EMPTY*
14. *IfStatement* → **if** (exp) statement | **if** (exp) statement **else** statement
15. *ForallStatement* → **forall** (exp in **ID**) statement
16. *outPutStatement* → **write** (**STRING**) | **writeln** (**STRING**) | **write** (**CHAR**) | **writeln** (**CHAR**) | **write** (**exp**) | **writeln** (**exp**)
17. *inputStatement* → **read** (**ID**)
18. *forStatement* → **for** (exp ; exp; exp) statement
19. aritExp → *exp***exp* | *exp* + *exp* | *exp* - *exp* | *exp*/*exp*
20. *RelExp* → *exp* rel *exp*
21. *callFuncStatement* → *ID* (callParams)
22. *callParams* → callParams-list
23. *callParams-list* → *callParams-list* , *callParam*
24. *callParam* → **Type ID**
25. *setExp* → *isSet* (**ID**) | **Add** (*exp* **IN** *termSet*) | **Remove** (*exp* **IN** *termSet*) | **Exists** (exp **In** *termSet*)
26. *termSet* → **ID** | *setExp*
27. *rel* → > | < | <= | >= | ! = | == | | &&
28. *terminal* → **ID** | **Integer** | **Float**

ID = letter(letter | digit)*
Integer = -?digit+
Float = -?digit*[.]digit+
elem = {integer}|{Float}
STRING = ¨"
CHAR = {letter}
TYPE = **Integer** | **Float** | **elem** | **set**
EMPTY = constante

letter = [a-zA-Z]
digit = [0-9]

O regex completo de todas as construções pode ser visto no arquivo skylang.l

Palavras reservadas da linguagem: write, writeln, add, exists, for, forall, remove, in, empty, if, else, float, elem, set, return, read.