Especificação da Etapa 4 do Projeto de Compilador

Análise Semântica

A quarta etapa do trabalho de implementação de um compilador para a **Linguagem Amarela** consiste nas primeiras verificações semânticas durante o processo de compilação. Estas verificações fazem parte do sistema de tipos da Linguagem **Amarela**, que tem como característica a existência de coerção entre os tipos inteiros, flutuantes e booleanos de acordo com um conjunto de regras detalhado na Seção 2 deste documento. Toda a verificação de tipos é feita de forma estática, em tempo de compilação, e deve considerar o nível de **escopo aninhado**. Finalmente, todos os nós da Árvore Sintática Abstrata (AST), gerada na etapa anterior, devem ter obrigatoriamente um campo que indica o seu tipo. O tipo de um determinado nó da AST pode, em algumas situações, não ser definido diretamente. Nestes casos, seu tipo deve ser inferido de acordo com as regras de inferência da linguagem (apresentadas na Seção 2). Uma série de testes de coerência comportamental das construções sintáticas reconhecidas e representadas na AST devem ser feitas nesta etapa. A Seção 1, a seguir, apresenta as funcionalidades necessárias que devem ser apresentadas nesta etapa. A Seção 3 apresenta os códigos de retorno para os possíveis erros semânticos encontrados.

1 Funcionalidades necessárias

Os resultados compreendem *a correção dos problemas encontrados na etapa anterior* e estas funcionalidades.

1.1 Escopo aninhado na linguagem Amarela

A gramática da linguagem permite que variáveis locais possam ser declaradas dentro de um bloco de código (delimitado por `{` e `}`). Uma árvore de tabela de símbolos deve ser criada durante o processo de análise sintática para permitir a verificação de escopo aninhado, começando pelo escopo global, passando pelo escopo das funções e enfim o escopo dos blocos (que podem conter outros blocos recursivamente). O encontro do tipo de um identificador deve acontecer através das regras de **escopo estático**, da seguinte forma: primeiro no escopo onde o identificador foi encontrado; em seguida, nos escopos acima considerando a árvore; em seguida, na função que contém a árvore de blocos; e, por fim, no escopo global. Por exemplo, considerando o código abaixo, o tipo da variável `var` deve ser procurado inicialmente no Bloco C, em seguida no Bloco A, em seguida na função f e por fim, caso ainda não foi encontrado, no escopo global. **Dica:** A forma mais simples de implementar escopo aninhado é através de uma pilha.

```
//Escopo global

int f ()
//Escopo da função
{
   {
      //Bloco A
      {
         //Bloco B
      };
      {
         //Bloco C
         var = 10;
      }
   }
}
```

1.2 Verificação de declarações

Todos os identificadores devem ter sido declarados no momento do seu uso, seja como variável, como vetor ou como função. Essa verificação de declaração prévia deve considerar o escopo aninhado da linguagem. Todas as entradas na tabela de símbolos devem ter um tipo associado conforme a declaração, verificando-se se não houve dupla declaração ou se o símbolo não foi declarado. Variáveis com o mesmo nome podem existir em escopos diferentes.

1.3 Uso correto de identificadores

O uso de identificadores deve ser compatível com sua declaração e com seu tipo. Variáveis somente podem ser usadas sem indexação, vetores somente podem ser utilizados com indexação, e funções apenas devem ser usadas como chamada de função, isto é, seguidas da lista de argumentos entre parênteses.

1.4 Tipos e tamanho dos dados

Uma declaração de variável deve permitir ao compilador definir o tipo e o tamanho (descrito na Seção 2) da variável na sua entrada na tabela de símbolos. Com o auxílio dessa informação, quando necessário, os tipos de dados corretos devem ser inferidos onde forem usados, em expressões aritméticas, relacionais, lógicas, ou para índices de vetores. Isso implica que todos os nós da AST são candidatos a terem um tipo definido de acordo com as regras de inferência de tipos. Esse processo de inferência está descrito na Seção 2.

1.5 Anotação da coerção de tipos

Os tipos inteiro, flutuante e booleanos podem sofrer coerção de acordo com o conjunto de regras apresentados na Seção 2 deste documento. A solução desta etapa deve marcar todos os nós da AST onde uma coerção deverá acontecer no momento da geração de código. Note que a coerção em si ainda não deve acontecer, apenas deve-se detectar e anotar na AST qual coerção deverá acontecer.

1.6 Argumentos e parâmetros

A lista de argumentos fornecidos em uma chamada de função deve ser verificada contra a lista de parâmetros formais na declaração da mesma função. Cada chamada de função deve prover um argumento para cada parâmetro, e ter o seu tipo compatível. A compatibilidade de tipos da linguagem é apresentada na Seção 2 deste documento.

1.7 Verificação de tipos em comandos

Todos os comandos simples da linguagem deve ser verificados semanticamente. O comando `input` somente aceita identificadores de qualquer tipo como parâmetro; o comando `output` aceita um literal `string` ou uma expressão aritmética a ser impressa. O comando de retorno `return` deve ser seguido obrigatoriamente por uma expressão cujo tipo é compatível com o tipo de retorno da função. Prevalece o tipo do identificador em um comando de atribuição.

2 Sistema de tipos da Linguagem Amarela

2.1 Coerção

As regras de coerção de tipos da Linguagem **Amarela** são as seguintes:

- Não há coerção para os tipos `string` e `char`
- Um tipo `int` pode ser convertido implicitamente para `float` e para `bool`
- Um tipo `bool` pode ser convertido implicitamente para `float` e para `int`
- Um tipo `float` pode ser convertido implicitamente para `int` e para `bool`, perdendo precisão

2.2 Inferência

As regras de inferência de tipos da Linguagem **Amarela** são as seguintes:

- A partir de `int` e `int`, infere-se `int`
- A partir de `float` e `float`, infere-se `float`
- A partir de `bool` e `bool`, infere-se `bool`
- A partir de `float` e `int`, infere-se `float`
- A partir de `bool` e `int`, infere-se `int`
- A partir de `bool` e `float`, infere-se `float`

2.3 Tamanho

O tamanho dos tipos da linguagem **Amarela** é definido da seguinte forma:

- Um `char` ocupa 1 byte
- Um `string` ocupa 1 byte para cada caractere
- Um `int` ocupa 4 bytes
- Um `float` ocupa 8 bytes
- Um `bool` ocupa 1 byte
- Um vetor ocupa o seu tamanho vezes o seu tipo

2.4 Código de tipos

Para simplificar a codificação do compilador, sugere-se a utilização das seguintes definições:

```
#define IKS_INT        1
#define IKS_FLOAT      2
#define IKS_CHAR       3
#define IKS_STRING     4
#define IKS_BOOL       5
```

3 Códigos de retorno para erros semânticos

A lista abaixo apresenta os códigos de retorno que devem ser utilizados quando o compilador encontrar erros semânticos. O programa deve lançar uma chamada exit utilizando esses códigos imediamente após a impressão da linha que descreve o erro encontrado. Outros erros podem ser criados pelo grupo, bastante para tal adicioná-los ao final desta lista, informando o professor da existência deles no momento da submissão.

```
#define IKS_SUCCESS            0 //caso não houver nenhum tipo de erro

/* Verificação de declarações */
#define IKS_ERROR_UNDECLARED  1 //identificador não declarado
#define IKS_ERROR_DECLARED    2 //identificador já declarado

/* Uso correto de identificadores */
#define IKS_ERROR_VARIABLE    3 //identificador deve ser utilizado como variável
#define IKS_ERROR_VECTOR      4 //identificador deve ser utilizado como vetor
#define IKS_ERROR_FUNCTION    5 //identificador deve ser utilizado como função

/* Tipos e tamanho de dados */
#define IKS_ERROR_WRONG_TYPE  6 //tipos incompatíveis
#define IKS_ERROR_STRING_TO_X 7 //coerção impossível do tipo string
#define IKS_ERROR_CHAR_TO_X   8 //coerção impossível do tipo char

/* Argumentos e parâmetros */
#define IKS_ERROR_MISSING_ARGS    9  //faltam argumentos
#define IKS_ERROR_EXCESS_ARGS     10 //sobram argumentos
#define IKS_ERROR_WRONG_TYPE_ARGS 11 //argumentos incompatíveis

/* Verificação de tipos em comandos */
#define IKS_ERROR_WRONG_PAR_INPUT  12 //parâmetro não é identificador
#define IKS_ERROR_WRONG_PAR_OUTPUT 13 //parâmetro não é literal string ou expressão
#define IKS_ERROR_WRONG_PAR_RETURN 14 //parâmetro não é expressão compatível com tipo do retorno
```

4 Requisitos Obrigatórios

Os requisitos obrigatórios são os mesmos da seção equivalente na especificação das etapas anteriores do projeto de compilador. O ambiente de compilação e execução desta etapa podem ser definidos da seguinte forma:

```
$ git pull origin master
$ cmake -DETAPA_1=OFF -DETAPA_2=OFF -DETAPA_3=OFF -DETAPA_4=ON ..
$ make
```

Para os testes da avaliação automática, basta executar o comando:

```
ctest -R e4
```

Caso os testes falhem, utilize a opção `-V` com o comando `ctest`.

5 Dúvidas

Crie um novo tópico no fórum de discussão do projeto de compilador no moodle.

Bom trabalho!