1 2  9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA

Projeto de Compiladores 2021/22

Compilador para a linguagem deiGo

1. Escrita da gramática da linguagem

Separou-se o símbolo Program em duas produções: uma delas terminal e outra que gera um símbolo não terminal Declarations.

O símbolo Declarations pode gerar uma produção VarDeclaration ou uma FuncDeclaration. Mas também existe a possibilidade de gerar ambas e chamar-se recursivamente pela esquerda.

Optou-se por agregar a produção do símbolo VarSpec ao símbolo VarDeclaration, desaparecendo este da gramática. Assim, o VarDeclaration ficou a englobar a combinação das suas produções com as do VarSpec. Por outro lado, para permitir declarações simultâneas de variáveis foi criado o símbolo auxiliar, Aux, que pode gerar uma produção terminal ou produção terminal antecedida de uma chamada recursiva à esquerda.

O símbolo Type não sofreu alterações.

Posteriormente, explicitou-se as produções do símbolo FuncDeclaration.

Na produção Parameters, as declarações simultâneas de parâmetros foram conseguidas através de um símbolo semelhante ao Aux, o Aux2.

O símbolo FuncBody não sofreu alterações.

De seguida, explicitou-se as produções do símbolo VarsAndStatements existentes e adicionou-se uma produção para recuperação local de erros.

No símbolo Statements evidenciamos as produções que originam if's, as produções que originam for's, as produções de return e as produções de print. Por motivos de implementação posterior da árvore acabamos por acrescentar uma produção que deriva do símbolo FuncInvocation diretamente no Statement. Posteriormente, foi utilizada uma produção com recursividade à esquerda para gerar produções Statement encadeadas, esta produção, Aux4, possuí duas produções de recuperação local de erros, uma delas terminal e outra que permite intercalar com outros Statements ou erros.

Em contrapartida, para o símbolo ParseArgs adicionou-se uma produção de recuperação local de erros.

Para o símbolo FuncInvocation adicionou-se uma produção de recuperação local de erros e retirou-se a produção anteriormente mencionada, que passou a produção direta da produção Statement.

No caso do símbolo Expr explicitou-se as produções existentes, adicionou-se uma produção de recuperação local de erros e acrescentou-se a produção proveniente da produção FuncInvocation. Para permitir uma sucessão de símbolos Expr recorreu-se a uma produção de lógica congruente com o Aux, o Aux3.

2. Algoritmos e estruturas de dados utilizados na AST e na tabela de símbolos

Optámos por uma árvore implementada através de vetores. Cada estrutura ast_node_t é composta por um const char * id que permite distinguir os nós, por um const char * type que será utilizado para guardar o tipo do nó resultante da análise semântica, um int temp que será utilizado posteriormente na geração de código, um vector_t * children onde serão guardados os nós filhos e um token_t * token onde serão guardadas informações sobre os tokens. Deste modo, foi construido a função add_child que adiciona um nó ao vetor children. Por sua vez a função add_as_siblings tem como parâmetros dois nós, parent e node, e tem como objetivo eliminar o nó node, passando os seus filhos a filhos do parent.

A estrutura vector_t é constituída por um int size que indica o número de elementos que o vetor contém, o int capacity que indica o número máximo de elementos do vetor e void ** data que contém os filhos do nó. Para este último, optou-se por um ponteiro void para reaproveitar a estrutura. O vector é redimensionado conforme a necessidade pela função de inserção push_back, ou seja, se o size igualar a capacity, esta é duplicada e a data realocada, sendo o elemento inserido.

A estrutura token_t é constituída por um char * value que contem informação particular ao token, int column e int line, que indicam a posição do token utilizada na impressão de erros.

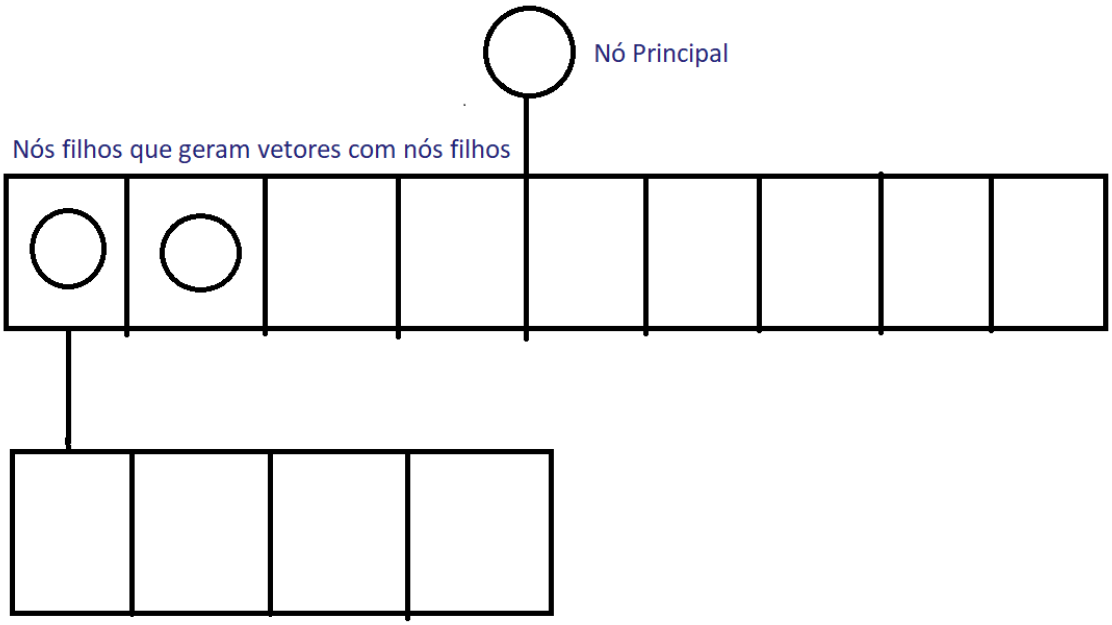
Para a tabela de símbolos utilizou-se um vetor, symbol_table, que armazena estruturas symbol_table_t e essas por sua vez contém estruturas symbol_t que representam os símbolos do programa.

A estrutura symbol_table_t contém um char * id que permite identificar as funções ou variáveis globais, um int is_defined que é uma flag para o controlo das definições de funções, um vetor, symbols, com os símbolos e um vetor de parâmetros no caso de ser uma função.

A estrutura symbol_t tem um char * id, para identificação, os int is_param, int is_func e in tis_used que são flags para a deteção de erros, um vetor_t parameters para simplificar a impressão na AST anotada, um char * type para anotar o tipo e um int temp para guardar o registo na geração de código.

O algoritmo percorria a árvore recursivamente sendo que alguns dos nós eram tratados iterativamente, nomeadamente, nós com número e posição dos filhos bem definida.

A imagem seguinte demonstra a estrutura da árvore implementada:



3. Geração de Código LLVM

Inicialmente a função generate_llvm começa por declarar as funções printf, atoi e constantes para imprimir os literais.

Seguidamente através da função declare_const_str recorre-se a uma primeira passagem recursiva na AST para declarar as constantes literais.

Posteriormente é efetuada uma passagem pelos nós declarativos e são declaradas funções e variáveis globais. Nas declarações, a função main é sempre gerada com os parâmetros argc e argv para permitir a utilização de parâmetros passados à função através da linha de comandos. Ao mesmo tempo, as variáveis são sempre inicializadas com valores default.

No corpo das funções percorre-se os nós e entra-se nas funções criadas para cada tipo de nó.

Na declaração de variáveis é alocada memória e os valores inicializados com o default, sendo o registo temporário guardado.

No assign procura-se a variável local e globalmente, utilizando-se a instrução store diretamente se for um literal ou desenvolvendo-se a expressão, sendo o seu resultado final guardado no registo da variável.

Desenvolveu-se expressões a partir de uma função recursiva que começa a avaliação pelos símbolos terminais e realiza cada operação guardando o seu valor na variável temp do nó. Os literais eram guardados recorrendo a uma operação de add para que todos os eles guardassem os valores e não ponteiros.

Ao longo das restantes funções foi utilizada a técnica de percorrer inicialmente a tabela de símbolos local, se não houver correspondência efetua-se uma segunda procura pela tabela de símbolos global, carregando-se o seu valor que posteriormente irá ser utilizado nas diversas operações.

Nas funções que geram if's é feita a comparação com a expressão e intercala-se a criação dos labels com a chamada da função que irá criar o corpo da mesma, assim irá ser permitido o encadeamento de if's.

Seguindo a mesma técnica utilizada nos if's foi criada a função que gera os for's.

Ao longo do código foram utilizadas funções auxiliares: para gerar o tipo de dados característicos do llvm, gerar os valores de default do mesmo, contar o número de caracteres normais, contar o número de caracteres de escape e imprimir os floats.

Inês Martins Marçal 2019215917

Noémia Quintano Mora Gonçalves 2019219433